

Keio University Master's Thesis Academic Year 2006

<p>LOLCAST: Abstract Layered Overlay Multicast Protocol</p>

Keio University Graduate School of Media and Governance

Kohei Ogura

LOLCAST: Abstract Layered Overlay Multicast Protocol

Summary

The Internet has become a social infrastructure, and peoples are using the Internet as a place to present their creation. These creations takes various media type such as text, audio and video. However, peoples are still not free to go for enjoying group communication using an real-time streaming contents. The difficulty lies in that both sender and receiver is just only an ordinary user. Objective of this research is to support creative activity of users, distributing real-time streaming contents to large group of peoples.

There are several group communication models proposed in the past. However, there are technical and policy issues to solve for targetting ordinary users. Recently, Overlay Multicast, a substitute technology for IP Multicast has been proposed which could support ordinary users. Nevertheless, Overlay Multicast has two major issues to consider, heterogeneous resource environment and unstableness at end nodes.

In this paper, LOLCAST is proposed as an adaptive Overlay Multicast protocol for real-time group communication in a heterogeneous environment to solve these issues. In LOLCAST, data is divided into multiple layers using abstract layered data structure. Number of layer is used for the main metric to construct the multicast tree to suffuse the demand for end nodes resource environment. Furthermore, multi-path layer distribution method for fast recovery from multicast tree partition and congestion avoidance method are proposed to deal with the instability at end node.

LOLCAST has been designed and implemented as an streaming application sending abstract layered data. In the evaluation, it has been confirmed that LOLCAST solved the major issues in Overlay Multicast research for realizing the objective of this research. Using LOLCAST, user could send and receive contents with ordinary resource environment and freely select media type or quality of the content on demand. Furthermore, LOLCAST prevents discontinuation of the content delivery in case of network congestion and multicast tree partition.

Keywords: 1.Overlay Multicast, 2. Content Distribution 3. Layered Coding

Keio University Graduate School of Media and Governance
Kohei Ogura

LOLCAST:
Abstract Layered Overlay Multicast Protocol

論文要旨

社会インフラとしてのインターネットは成熟期を迎え、人々の活動を支える表現の場として活発に利用されるようになった。この新たな表現の場において、人々は様々な表現メディアを用いて活動成果を発信している。しかし、現状においてこれらの人々がリアルタイム性の高い映像表現を用いた配信を行なうことは困難である。これは、受信者・配信者が共に一般利用者であり、計算機資源やネットワーク帯域資源に限界があることに起因する。本研究では、「多くの受信者を対象とするコンテンツ配信を行なう人々の表現活動を支援すること」を目的とする。インターネットにおいて放送型の通信を実現する技術はこれまでも盛んに研究が行なわれてきた。しかしこれらの技術は、一般利用者が手軽に利用できるものではない。近年、IP マルチキャストの代替手段として登場したオーバーレイ・マルチキャスト技術は一般利用者を対象とできるが、各利用者における資源環境の異種性への対応と不安定なエンドノードにより構築される通信基盤の維持の大きな二つの問題がある。

本研究では上述した問題点を解決するため、新たなオーバーレイマルチキャストプロトコルである LOLCAST を提案する。LOLCAST は各利用者における資源環境の異種性に適応するための機能として、抽象化されたレイヤを持つデータを利用する。これを配信するために、レイヤ数を基準としたマルチキャストツリーの構成を行う。さらに不安定な通信基盤の維持のために複数パスを利用した冗長的なレイヤ配信を行う機能とレイヤ構造の特徴を活かした輻輳制御の機能を提供する。

更に本研究では、LOLCAST を利用した抽象化されたレイヤ構造を持つデータのストリーミングを行うアプリケーションの設計・実装を行った。また評価として、実装を行ったアプリケーションによる実験、他プロトコルとの機能比較、プロトコルの処理速度の計測を行い LOLCAST がオーバーレイマルチキャスト研究における二つの問題点を解決したことを確認した。

LOLCAST により配信者は一般利用者が持ちうる現実的な資源環境で配信網を構築でき、受信者はその要求に基づいた自由なコンテンツの品質や表現の選択が行える。また、受信者は配信元の離脱やネットワークの輻輳状態においてもシームレスにコンテンツを視聴する事が可能となる。

キーワード: 1. オーバーレイマルチキャスト, 2. コンテンツ配信 3. 階層符号化

慶應義塾大学大学院 政策・メディア研究科
小椋康平

Acknowledgments

I would like to thank my thesis supervisors, Professor Jun Murai, Professor Osamu Nakamura, and Dr. Hideaki Imaizumi for their guidance and advice throughout the process of writing this thesis. Especially I would like to thank Dr. Hideaki Imaizumi for assinting me for writing this thesis. He guided me from the really fundamental stuffs such as “what is research” when I was working on my bachelor thesis. His request for the work was very hard for me but precise. His guidance always made me the research fun and excitng.

A huge thanks to all the members of the SING/IA* research group for assisting me for this research. I am especially grateful to Masaki Minami and Yasuhiro Ohara for supporting on both technical and mental stuffs. Support from Shin Shirahata, Masayoshi Mizutani, Yusuke Okumura, Akira Kanai, Yohei Kuga, Takaaki Ozaki, Ryu Sato and Toshiaki Hatano really helped for finish writing this thesis.

I am also thanks to my colleagues encouraging me for working together on a this hard task, Ryusaburo Tani, Masahumi Yoshida, Kazuhisa Matsuzono, Yoshihiro Toyama and Manabu Tukada. It was not the same without these members encouraging each other to reach the goal.

Above all, I would like to thanks to my family for the huge support to be here.

Contents

1	Introduction	1
1.1	Background	1
1.2	Objectives	1
1.3	Fundamental Propositions	2
1.4	Organization of This Paper	3
2	Current Group Communication Models and Issues	4
2.1	Server-client model	4
2.2	CDN	5
2.3	IP Multicast	5
2.4	Overlay Multicast	6
2.4.1	Topology classification in Overlay Multicast	7
2.4.2	Summary	7
2.5	Issues in Overlay Multicast	8
2.5.1	Approaches to adapt end node heterogeneity	8
2.5.2	Approaches to adapt end node instability	10
2.6	Summary	12
3	Overview of LOLCAST	13
3.1	Abstract layered data structure	13
3.2	Definitions used in LOLCAST	14
3.3	Tree structure of LOLCAST	15
3.4	Recovery method from node failure	16
3.4.1	Multi-path layer distribution	16
3.4.2	Congestion avoidance	17
4	Design of LOLCAST	19
4.1	Tree Parameters	19
4.2	Node Parameters	20
4.2.1	Fundamental Parameters	20
4.2.2	Source node specific parameters	21

CONTENTS

4.2.3	Relay node specific parameters	22
4.2.4	Node Initialization	22
4.3	Node State	24
4.4	Messages	25
4.4.1	Join Request Message	25
4.4.2	Join Accept Message	29
4.4.3	Notify New Parent Message	30
4.4.4	Notify Accepted Message	30
4.5	Join Procedure	31
4.5.1	Message passing in Join Procedure	31
4.5.2	Example of Join Procedure	31
4.6	Leave Procedure	34
4.6.1	Message passing in Leave Procedure	34
4.6.2	Example of Leave Procedure	34
5	Implementation of LOLCAST	37
5.1	Implementation Environment	37
5.2	Implementation Overview	38
5.3	Protocol Processing Module	39
5.3.1	Data Structure	40
5.3.2	Message Format and Message Handling Methods	42
5.3.3	Tree Handling Methods	45
5.4	Application Module	50
5.5	Network Module	51
5.6	Simulation Module	51
5.7	User Interface	51
5.7.1	Application Mode	52
5.7.2	Simulator Mode	52
6	Evaluation	54
6.1	Verification of Protocol Process	54
6.1.1	Experimental Setup	55
6.1.2	Experiment Result	57
6.2	Functional Comparison	60
6.3	Performance of protocol process	62
6.3.1	Environment for performance evaluation	62
6.3.2	Parameters for performance evaluation	63
6.3.3	Measurement procedure	63
6.3.4	Measurement Result for Join Procedure	63
6.3.5	Measurement Result for Leave Procedure	64
6.4	Summary	66

CONTENTS

7	Conclusion and Future Work	68
7.1	Conclusion	68
7.2	Future Work	69

List of Figures

1.1	Quality control in existing streaming service	3
2.1	Example of server-client model	5
2.2	Problems in IP Multicast	6
2.3	Difference between Overlay Multicast and IP Multicast	6
2.4	Control and Data Topology	7
2.5	Comparison between multiple version and multiple layer ap- proach	9
2.6	Redundant path in control topology	10
3.1	Layered structure in LOLCAST	13
3.2	Multicast tree in LOLCAST	15
3.3	Multi-path layer distribution	16
3.4	Congestion control	17
4.1	lowerLayerNodes()	21
4.2	minDepthNode()	22
4.3	linkUp()	23
4.4	linkDown()	23
4.5	State diagram for node	24
4.6	State diagram for node in tree structure	24
4.7	extractPCS()	25
4.8	JoinRequest()	27
4.9	JoinAccept()	29
4.10	NotifyNewParent()	30
4.11	NotifyAccepted()	30
4.12	Message passing in Join Procedure	31
4.13	Join Procedure 1	32
4.14	Join Procedure 2	33
4.15	Join Procedure 3	33
4.16	Message passing in Leave Procedure	34

LIST OF FIGURES

4.17	Leave Procedure 1	35
4.18	Leave Procedure 2	35
5.1	LOLCAST Application Modules	38
5.2	nodeInfo Structure	40
5.3	treeNodeInfo Structure	40
5.4	layerInfo Structure	41
5.5	Data structure maintained by source node	41
5.6	Data structure maintained by relay node	42
5.7	Message Format	43
5.8	Message Handling Methods	44
5.9	Tree Handling Methods	45
5.10	generatePCS()	47
5.11	generatePCSRejoin()	49
5.12	Sample telnet commands in source node	50
5.13	Sample telnet commands in relay node	50
5.14	Running lolcast_app in Application Mode	52
5.15	Prompt for asking number of layers to request	53
5.16	Running lolcast_app in Simulation Mode	53
6.1	Network Topology of the experiment	56
6.2	Experimental Data	57
6.3	Generated LOLCAST Tree	58
6.4	Screenshot at Relay Node A	59
6.5	Join Redirect message received at relay node E	59
6.6	Process time for Join Procedure (4 layers/random layer/random leave)	65
6.7	Process time for Leave Procedure (4 layers/random layer/random leave)	66

List of Tables

5.1	Environment for implementation	37
6.1	Environment for experiment	55
6.2	Parameters set for experiment	57
6.3	Functional comparison of Overlay Multicast Protocols	60
6.4	Hardware and software environment	62
6.5	Parameters set for performance evaluation	63
6.6	Average process time for Join Procedure	64
6.7	Average process time for Leave Procedure	67

Chapter 1

Introduction

1.1 Background

The Internet has become a social infrastructure, and millions of peoples are working on this new infrastructure. These peoples are performing various creative activities such as self-produced art, music, movies, etc. due to the rich environment at end user on creating high-quality multimedia contents. In this research we state an individual or a group formed by their interest and performing creative activities on the Internet as Internet Community.

Internet Communities are using the Internet as a place to present their creation. These creations are made in various media type: text media such as novel and diary, audio media such as music and radio program, video media such as movie and comedy show. In addition, in the future it is unsurprising that there will be an creative activity using a new media type such as three-dimensional videos [1].

1.2 Objectives

As stated in Section 1.1, members of a Internet Community has an rich environment for creating high-quality multimedia contents. However, members of a Internet Community are still not free to go for enjoying neither broadcast self-produced real-time comedy show nor videoconferencing with large group of peoples which is a group communication using an real-time streaming contents. This problems difficulty lies in that both sender and receiver of the community member is just only an end user.

These user has three major characteristics to consider for realizing such communication. First is that user has no special equipment or financial support for sending the contents. Second is that user has a limitation in

computation and network resource for sending and receiving the contents. Last is each user has a heterogeneous resource environment for receiving the contents. In this paper we state such kind of user as ordinary user.

Our research objective is to support creative activity of the Internet Community, distributing real-time streaming contents to large group of peoples.

This research proposes an adaptive overlay multicast protocol in a heterogeneous environment for group communication such as real-time video streaming, LOLCAST (Abstract Layered Overlay multiCAST) to solve these issues. LOLCAST, assumes the multicast group size as several hundreds. LOLCAST uses layered coding and abstract layered data structure to adapt end nodes heterogeneity. In addition, this paper introduces fast recovery method from multicast tree partition and congestion control method using the characteristic of abstract layered data structure.

1.3 Fundamental Propositions

To achieve the goal stated in Section 1.2, there are three fundamental propositions to satisfy. In this section, each of the proposition will be described briefly.

1. User could send and receive contents with ordinary resource environment

Even if the content sender is an ordinary user, network for the content delivery can be constructed and maintained with a ordinary resource environment. Resource environment includes network and computing resource. Further out, there should not be a limitation for the receiver to receive the content. This should be concerned from the reason that ordinary users has a limited resource for sending and receiving contents.

2. User could freely select media type or quality of the content on demand

Every receiver can freely select media type or quality of the content within the bounds of their resource environment. Receivers request for the quality or media type of the content is various. This request frequently changes from users resource environment or interest level to the content.

For example Figure 1.1 illustrates a function of existing streaming services to select content quality. A website for streaming movie trailers [2] is offering three different quality for one content, “Mid”, “High” and “Fiber”, which is abstracting the network line of the user. In comparison, an Internet radio streaming service is offering “24kbps”, “56kbps”



Figure 1.1: Quality control in existing streaming service

and “96kbps” of audio stream data, which shows the bandwidth of the content. As described above, it is fundamental for the content sender to serve multiple quality or multiple media type for adapting to receivers resource environment or interest level, and receiver should be able to freely select this.

3. Seamless content delivery

User can receive the content from the sender seamlessly. Streaming media is a time-series data, which have a large affect on discontinuity of the data. For the reason of the discontinuity, disconnection of the sender or network congestion is thinkable.

1.4 Organization of This Paper

The remaining of the paper is organized as below. In Chapter2, issues of recent group communication methods are described in detail. Chapter3 covers overview of proposed new Overlay Multicast protocol LOLCAST, along with each distinctive functions to satisfy the fundamental propositions stated in Section 1.3. In Chapter4, protocol design of LOLCAST is described in detail, continuing on with each important procedure done in LOLCAST. The results through simulation analysis is shown in Chapter6, followed by the conclusion and future work in Chapter7.

Chapter 2

Current Group Communication Models and Issues

This section represents brief interpretation of existing models for group communication. Server-client model, CDN, IP Multicast and Overlay Multicast is discussed. For each method, issues to meet our research objective stated in Section 1.2 is mentioned.

2.1 Server-client model

Simple approach to realize group communication on the Internet is the server-client model. Server-client model does not require particular delivering method and therefore it is adopted in many real-time video streaming services.

Figure 2.1 shows that the data flow in server-client model is consist of multiple unicast streams sent from sender to each receiver. In such model, network bandwidth in a single link will be burdened. From this reason, server-client model has a difficulty for supporting ordinary user which has limited resource environment. In Figure 2.1, sender node is delivering data to receiver node *A*, *B* and *C*. Senders network resource is already exhausted by delivering to these three nodes. In this case, a new receiver node *D* is unable to retrieve the data.

As above, streaming service using a server-client model requires network bandwidth proportional to the group size. Consequently, streaming service using server-client model provided by an ordinary user is in very small size or using very limited quality of data.

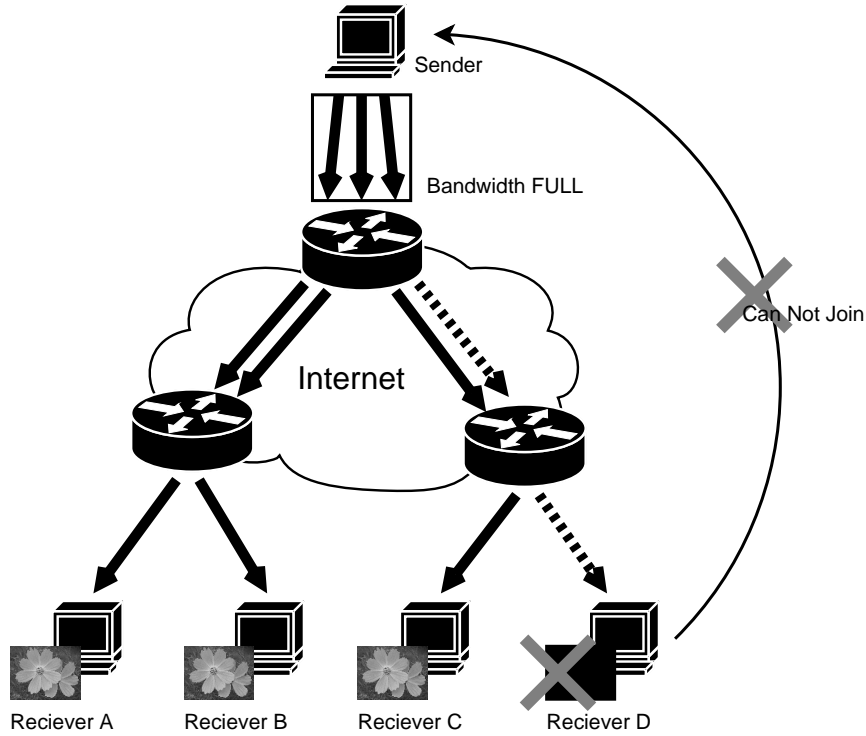


Figure 2.1: Example of server-client model

2.2 CDN

CDN (Contents Distribution Network) is realized by distributing the sender of Server-client model in multiple location described in Section 2.1. By distributing the sender into multiple location, which is the bottle neck, load for the sender decreases. In detail, the content is cached to the servers located in each Internet service provider, and sender reports to the receiver which server to join. However, there is no CDN service which ordinary user could use freely. In addition, it is assumed that every Internet service provider is joining to the CDN, and which could not support widespread users which is concerned.

2.3 IP Multicast

IP Multicast is the traditional method for group communication over the Internet. Though long time has elapsed since IP Multicast was initially proposed [3], IP Multicast still has both technical and policy issues such as inter-domain routing, diffusion of multicast capable routers, multicast

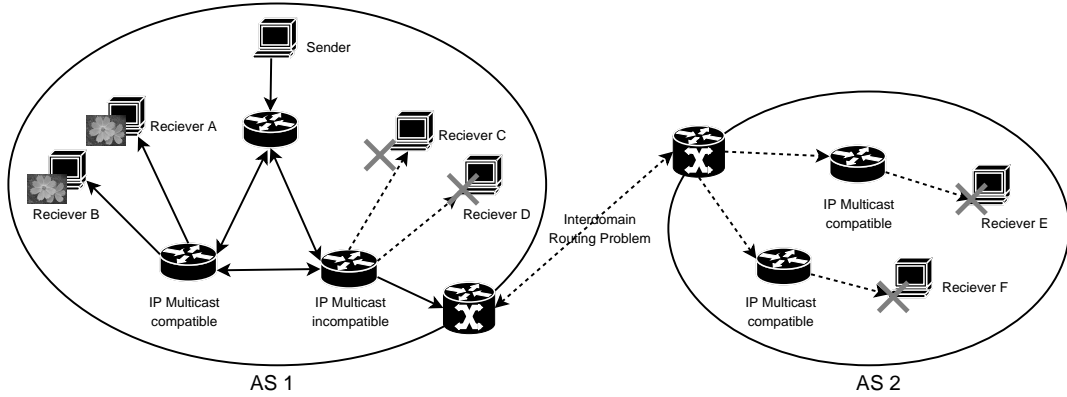


Figure 2.2: Problems in IP Multicast

address allocation, etc. to solve for wide area deployment [4].

Figure 2.2 shows an example of sender delivering the data to *A*, *B*, *C*, *D* and *E* using IP Multicast. In this case, receiver *C* and *D* are impossible to receive the data, because both receivers are connected to a IP Multicast incapable router. Furthermore, receiver *D* and *F* also could not retrieve the data due to the policy issues between *AS1* and *AS2*.

Therefore, ordinary user could not use IP Multicast as a method for group communication due to the widespread and distributed receivers. Presently use of IP Multicast is limited only in a sparse domain.

2.4 Overlay Multicast

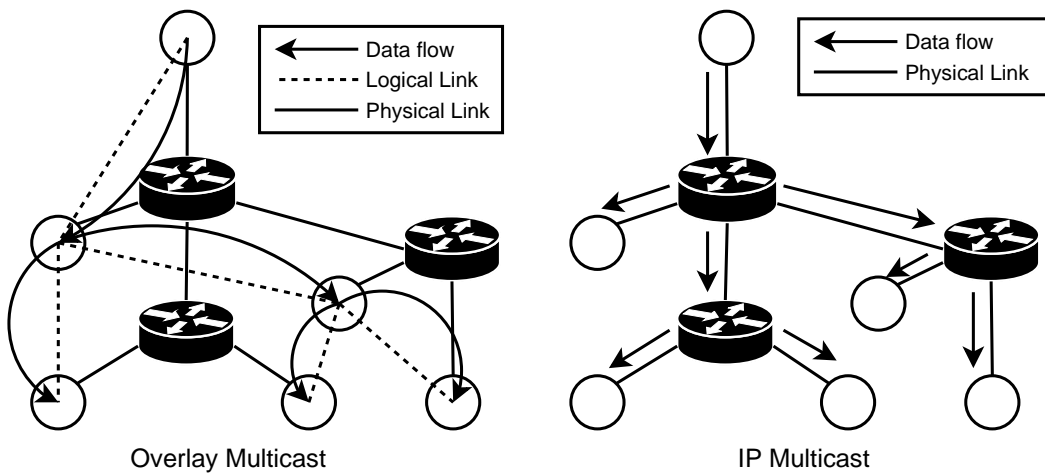


Figure 2.3: Difference between Overlay Multicast and IP Multicast

Recently, Overlay Multicast, a substitute technology for IP Multicast has become a hot topic for researchers. The basic idea of Overlay Multicast is to delegate multicast functionality such as, data replication, group management, multicast routing from IP layer to upper layer, mostly the application layer. Overlay Multicast constructs a logical network over the underlying IP network and use it as a infrastructure for multicasting. In Overlay Multicast, multicast is done by unicast between the nodes joining to the multicast group. Figure 2.3 illustrates how each method bear the function of data replication, which is routers in IP Multicast and end nodes in Overlay Multicast. Since the idea of Overlay Multicast first appeared [5], number of Overlay Multicast routing protocols has been proposed [6, 7, 5, 8, 9, 10, 11, 12, 13, 14, 15, 16, 17, 18, 19, 20].

2.4.1 Topology classification in Overlay Multicast

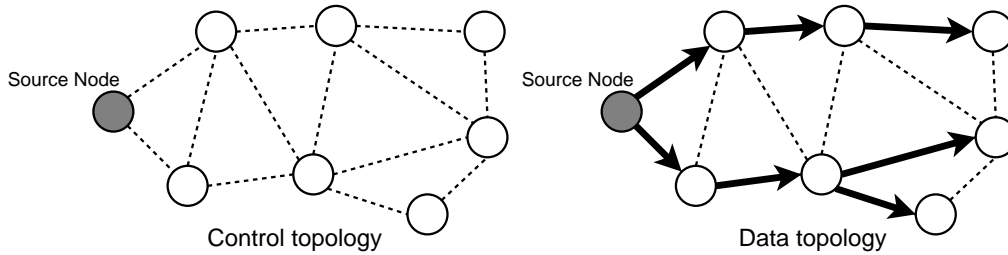


Figure 2.4: Control and Data Topology

Topology constructed by Overlay Multicast Protocol is categorized into two kinds, control topology and data topology [21]. Figure 2.4 illustrates an example of each topology. Control topology is used to manage node state and node information used in the protocol. Informations maintained by control topology is used in such as to run join and leave procedure or recovering multicast tree partition. Data topology is usually a subset of control topology. Over the control topology, data topology set the actual data flow between each node.

2.4.2 Summary

Overlay Multicast is realized by constructing a overlay network over the IP network formed by end nodes. Data is relayed between each end nodes and sender should send only at least one stream to distribute the content to large group of users. This satisfies the first fundamental proposition stated in Section 1.3 which is “User could send and receive contents with ordinary resource environment”.

2.5 Issues in Overlay Multicast

By delegating the multicast functionality to application layer, Overlay Multicast network relies on end nodes. This means every end node joining to the multicast group constructs and maintains the multicast tree to deliver the data. In such environment, Overlay Multicast research has two major issues to consider, end node heterogeneity and end node instability. These two issues are corresponding to the left two fundamental propositions stated in Section 1.3.

Each end node joining the multicast tree has a heterogeneous resource environment such as link bandwidth and computing resource. Each node will request different quality or media-type of contents to satisfy their resource constraint. Multicast method should handle this request flexibly. At the same time, multicast method should not limit the node to join by its resource constraint.

Unstability at an end node also should be considered for constructing stable multicast tree. A node failure will cause multicast tree partition, which stops the data transmission. From this reason, fast recovery method of multicast tree is required for reliable multicast tree.

This section illustrates recent approaches in both adapting end node heterogeneity and end node instability. Example of approaches taken in recent Overlay Multicast protocol is discussed briefly. In addition, remaining issues to meet our research objective is stated.

2.5.1 Approaches to adapt end node heterogeneity

There are mainly two approaches proposed to solve this difficulty. Primary method is multiple version approach and the secondary method is multiple layer approach. Figure 2.5 illustrates comparison between data structure used in both approaches. Example of data structure supports four different qualities of data.

Multiple version approach

In multiple version approach, multiple video data containing different quality and bit-rate for a single content (from low quality to very high quality in Figure 2.5) is sent by the source node. Receiver node selects the stream which suits their resource environment, especially the network bandwidth. This approach is taken by End System Multicast [5].

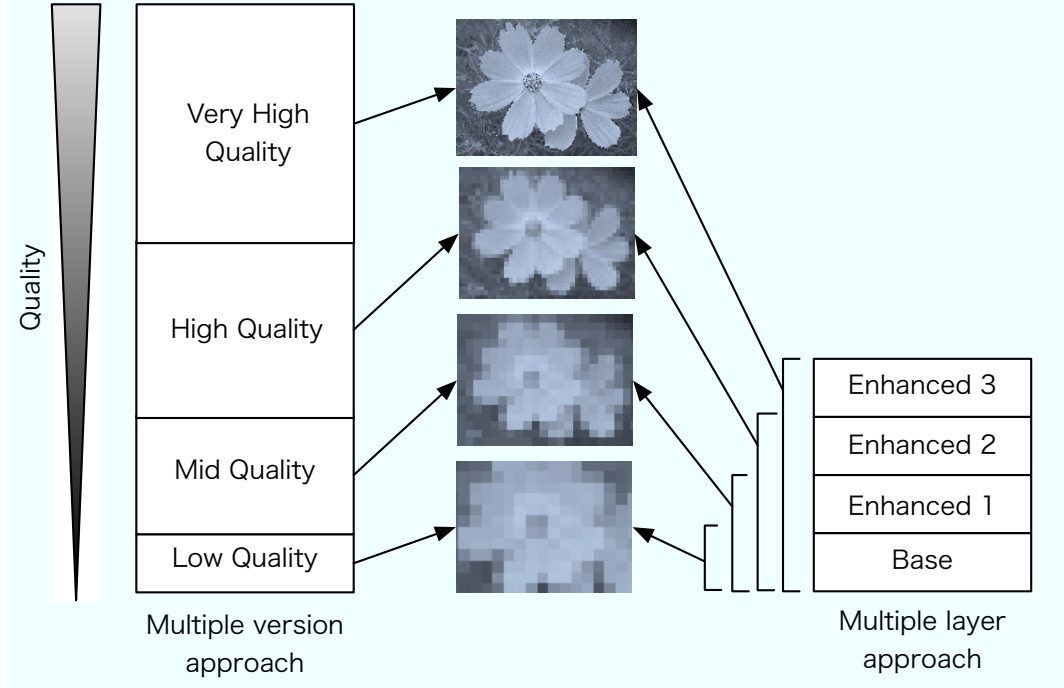


Figure 2.5: Comparison between multiple version and multiple layer approach

Multiple layer approach

In multiple layer approach, layered video coding is the key technology. In layered coding, video data is divided into multiple layers, as illustrated in Figure 2.5. Data included in each layer are non-overlapping each other. Layers are categorized into base layer and enhancement layer. Base layer provides the minimum quality of original video data, and it is fundamental for decoding the other layers. Enhancement layer provides additional data which improves video quality. Each layer has a dependency with layer directly below for decoding. Several layered coding method have been proposed, including MPEG-2 scalable profile [22], MPEG-4 scalable profile [23], H.263+ [24], MDC (Multiple-Description Coding) [25].

Multiple layer approach uses layered encoded data to adapt to heterogeneity. Source node sends the segmentalized video data with full layers, and receiver node acquires number of layers to sustain their resource environment. This approach is taken by Okada's work [14], Koguchi's work [26], LION [18], PALS [19] and our previous work [17].

Advantages and drawbacks

As referred as above, multiple layer approach uses data structure consisting of multiple layer which quality improves by increasing the number of layer. Multiple layer approach has an advantage in network bandwidth utilization compared to multiple version approach. Compared to multiple layer approach, multiple version approach needs a separate and overlapping data to support each quality, which burden the network bandwidth. Another advantage for multiple layer approach is that it could handle wide-range of requests for quality very flexibly by just increasing the number of layer. One drawback for multiple layer approach is that layered coding uses complicated encoding method which requires some computing resource.

2.5.2 Approaches to adapt end node instability

Proposed approaches for adapting instability of end node can be classified in to two types [27]: reactive approach and proactive approach. Reactive approach start tree restoration process after the node detects parent node failure. Mainly this approach runs join procedure from the beginning, therefore it takes long time to rejoin to the tree. This approach is taken by Narada [5, 8], Koguchi's work [26], LION [18] and PALS [19]. In contrast proactive approach deal with the node failure before it happens. This approach is taken by Okada's work [14], HostCast [15], Yang's work [27] and PRM [20]. Each of the function by proactive approach is illustrated briefly.

Redundant control path in HostCast

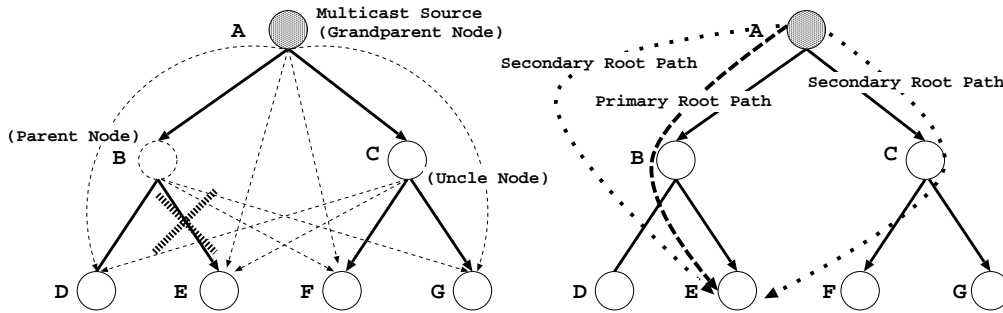


Figure 2.6: Redundant path in control topology

Idea of HostCast is to accurately measure the overlay path condition and help the group member to find a new parent node quickly when its original parent node is lost. Figure 2.6 illustrates the idea of HostCast to construct

a redundant path in control topology. The example shows how node E recovers from multicast tree partition. In normal state, path $A - B - E$ is used to deliver the data to node E . At the same time, node E maintains secondary parents in the control topology, grandparent node A and uncle node C . These links are called secondary root path. If parent node B leaves from the multicast tree, tree will be partitioned between $B - E$. Soon as node E detects node B leaved, it sends join request message to random secondary parents using the control path connected previously. By joining to another parent node, node E recovers to the multicast tree. By preparing the redundant path in advance in the control topology, node could find new parent node in advance. However, it still requires process time for switching the control topology and also to request new parent for the data by only setting paths on control topology.

Parent candidate list backup in Okada's work

Another method for adapting instability of end node is proposed in Okada's work [14]. Idea of Okada's work is to backup the list of parent candidates received in the join phase to find the new parent quickly. The node does not have to run the full join procedure but only has to send join request to the node in parent candidate list. However the saved parent candidates is possible for becoming invalid, due to the leave of parent candidate node. In addition it requires process time until the node start receiving the data due to the same reason in HostCast.

Pre-computing backup nodes in Yang's work

Yang's work [27] focuses on this issue and studied it for the main feature of proposed protocol. Main idea is that nodes primary parent pre-computes the parent candidate for each of its children. In case of node failure, node rejoins to pre-computed parent node immediately. Node should send only one message is required to rejoin to the data topology and start receiving the data. Nevertheless, it still has a time for data lost until it adds a path on the control topology.

Randomized data forwarding in PRM

PRM [20] also takes proactive approach, but it differ from other proactive approaches by it uses data topology for realizing the function. PRM uses a randomized forwarding method which every node chooses a number of other node uniformly at random. Every chosen nodes forwards the data to the node with a low probability. The recovery time for this method is very small

by redundantly receiving the data from number of parent nodes. However, the traffic for sending overlapping data can be very large in a case such as live video streaming which is our target application.

2.6 Summary

This chapter illustrated the current group communication models and issues. There are several models proposed for group communication such as server-client model, CDN, IP Multicast and Overlay Multicast. Server-client model, CDN and IP Multicast has a both technical and policy issues left to satisfy the first fundamental proposition, “User could send and receive contents with ordinary resource environment”. Recently Overlay Multicast has been proposed for the substitute technology with IP Multicast. Overlay Multicast technology satisfies the first fundamental proposition by constructing a logical network over the underlying IP network and use it as a infrastructure for multicasting. However, Overlay Multicast has two major issues to satisfy the left two fundamental propositions: function to adapt end node heterogeneity and functions to adapt end node instability.

Chapter 3

Overview of LOLCAST

In order to solve the issues illustrated in Chapter 2, this paper proposes a novel Overlay Multicast protocol LOLCAST. First overview of the protocol is illustrated. Next, describes the data structure and definitions used in LOLCAST. Next, example of tree structure constructed by LOLCAST is illustrated. Last, this section describes recovery method from node failure and congestion avoidance.

3.1 Abstract layered data structure

The basic idea of layered data structure is similar to multiple layer approach, which data consists of multiple layers to support wide-range of information amount. Layered data structure used in LOLCAST basically inherits multiple layer approach but can support not only layered coded data but also combined data from various type of data abstractly. Base layer and enhancement layer will be used for the term to describe each layer in LOLCAST.

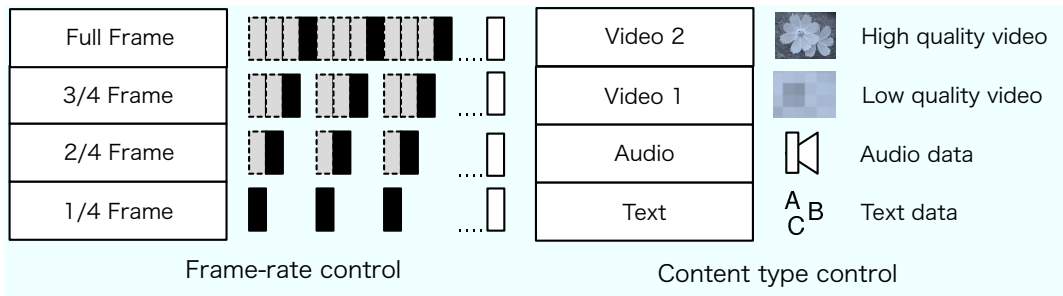


Figure 3.1: Layered structure in LOLCAST

Several usage of layered data structure can be conceivable which is illustrated in Figure 3.1. Left figure illustrates a structure for controlling the

frame-rate of video stream. Each layer carries each frame of video stream in such as DV format. Base layer offers 1/4 frame of full-frame video stream, and by increasing the layer, frame-rate increases. Data with full layer offers full-frame of video data. Right figure illustrate a structure for streaming various type of content format. Base layer offers text data, and each enhancement layer increases the amount of information using audio and video data.

3.2 Definitions used in LOLCAST

In this section, we introduce the following definitions used in LOLCAST. These definitions will be used for explaining the functions of LOLCAST.

- **Layer encoded data** $\{l_0, l_1, l_2... l_n\}$
This definition stands for each layered coded data. l_0 is the base layer and the rest are enhancement layer. l_n is the top layer which original data carries, sent by source node.
- **Nodes** $\{N_0, N_1, N_2... N_n\}$
This definition stands for the node which joining to the multicast tree. N_0 is the source node. n is the total number of nodes joining to the multicast tree.
- **Number of layer** $\{L_0, L_1, L_2... L_n\}$
Number of layer is the layers which source node maintains or certain node requests. L_0 is the maximum number of layer which the data carries sent by source node. For example, if source node carries layered coded data with 5 layer, $L_0 = 5$, and N_0 has layer l_0 through l_5 .
- **Number of child nodes** $\{C_0, C_1, C_2... C_n\}$
This definition stand for the number of children nodes, which receiving the stream from certain node. Each node sets a maximum number of children nodes to support, according to its own network bandwidth represented as c_{max} . In addition, source node should set a minimum number of children nodes for every node joining to multicast tree, which will be represented as c_{min} . c_{min} should be larger than one to construct a tree and node should set c_{max} larger than c_{min} .
- **Depth** $\{D_0, D_1, D_2... D_n\}$
Depth shows the nodes position in the multicast tree. Source node N_0 is the top node in multicast tree, therefore $D_0 = 0$. If N_j has two ancestor nodes between the path to source node, $D_j = 2$.

3.3 Tree structure of LOLCAST

Characteristic of layered coding used in multiple layer approach makes a restriction in the method for delivering the data. The point is that there are dependencies between each layer. First enhancement layer l_1 requires the base layer l_0 for decoding. Second enhancement layer l_2 needs both primal enhancement layer l_1 and the base layer l_0 for decoding, and so on. This means each layer could not be sent apart to decode the data.

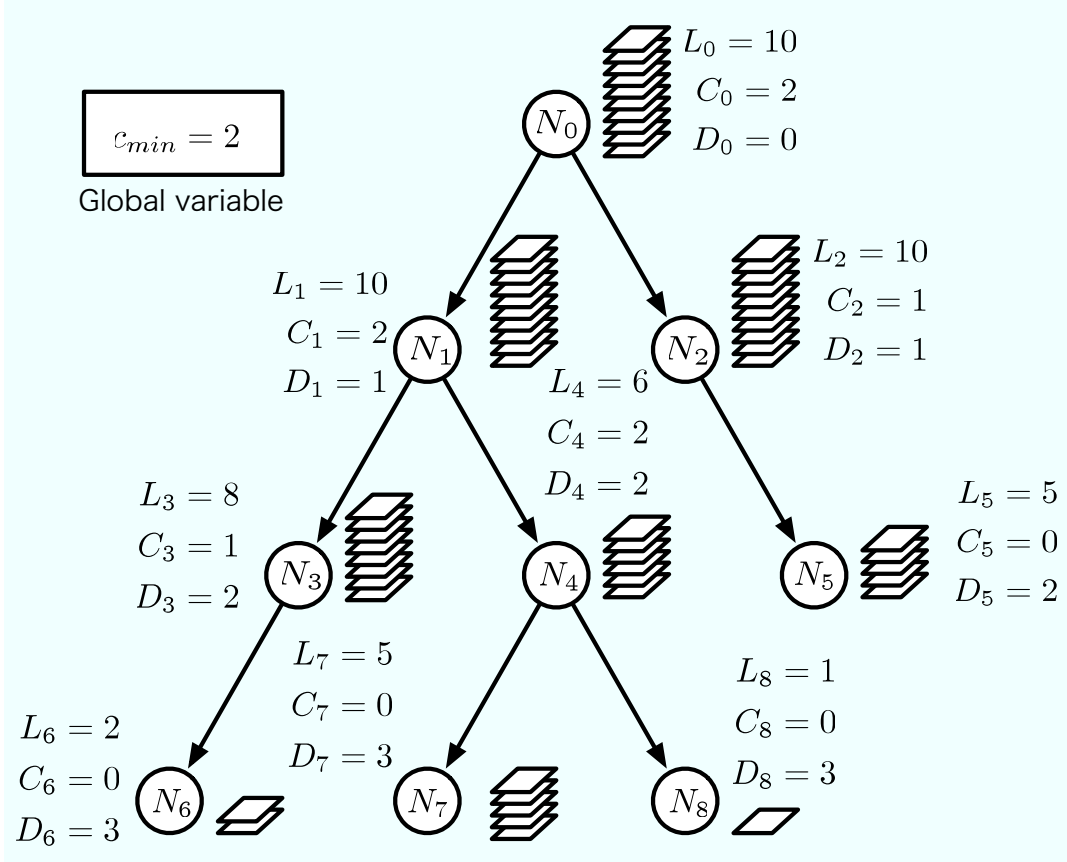


Figure 3.2: Multicast tree in LOLCAST

Therefore, LOLCAST uses number of layers which each node requests as the metric to construct the multicast tree. Target group size of LOLCAST is several hundreds at maximum.

There are three types of nodes in LOLCAST, source node, relay node and leaf node. In LOLCAST, source node maintains the entire multicast tree structure and serves the original data transmitted over the multicast tree. Relay and leaf node requests source node for the proper parent node to

join, and receives the data it requests. Leaf node only receives the data and does not have any child node.

Example of multicast tree constructed by LOLCAST is illustrated in Figure 3.2. This tree has minimum children nodes for two ($c_{min} = 2$) and maximum number of layer for ten ($L_0 = 10$). For simplicity, c_{max} for every node is set to 2. For example, node N_4 is receiving data consist of six layers ($l_0...l_5$) from its parent node N_1 . In this case, node N_4 could serve any child node requesting not more than six layers, which is node N_7 and N_8 in Figure 3.2. Node N_6 and N_8 are leaf nodes requesting one or two layers. Leaf node N_6 and N_8 can be expected as a node with very small resource environment, such as wireless devices.

3.4 Recovery method from node failure

Overlay Multicast relies on unstable infrastructure, compared to IP Multicast. From this reason, researchers have large attention in the method for handling instability of end node. Especially, video streaming requires fast recovery to avoid information lost, which is our target. This section introduces functions to handle this issue. First is multi-path layer distribution method and second is congestion avoidance method.

3.4.1 Multi-path layer distribution

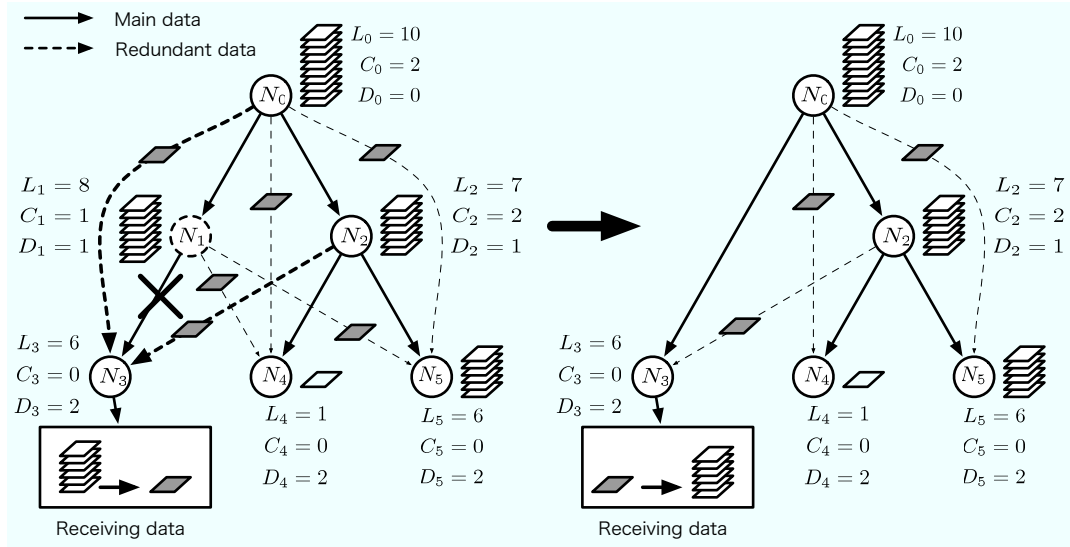


Figure 3.3: Multi-path layer distribution

This section describes multi-path layer distribution method for fast recovery in case of node failure. There are several methods for fast recovery from multicast tree partition. HostCast [15] uses redundant path from source node in control topology to shorten the time after detecting the node departure and to restart sending data. However this approach requires time to converge the multicast tree before recovery. LOLCAST uses data topology to construct a redundant data delivery path compared to HostCast. By directly sending redundant data from multiple parent nodes, recovery time shortens compared with other methods.

Figure 3.3 illustrates how node N_3 recovers when parent node N_1 failed. In normal state, path $N_0-N_1-N_3$ is used to deliver data from source node N_0 to node N_3 . At the same time node N_3 is redundantly receiving base layer from node N_0 and N_2 . In addition, all nodes has the option to request number of layers for redundant data, alternative for using base layer. When node N_1 fails from the multicast tree, path $N_0-N_1-N_3$ becomes unavailable. As soon as node N_3 detects his parent node N_1 has failed, node N_3 switch to the redundant data receiving from N_0 or N_2 to reduce the information lost. While receiving the redundant data from N_0 or N_2 , N_3 recovers into multicast tree by finding a new parent node N_0 and starts receiving data with the requesting quality.

3.4.2 Congestion avoidance

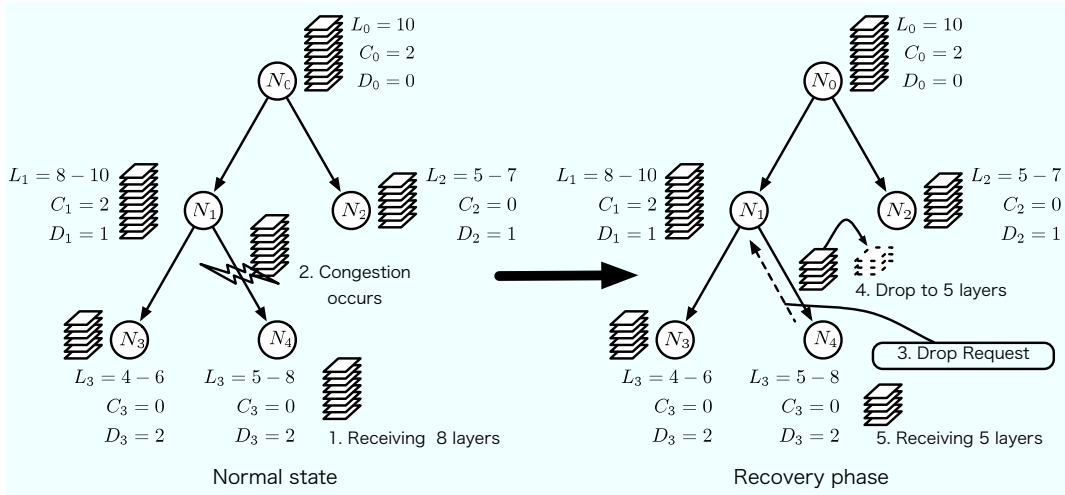


Figure 3.4: Congestion control

Figure 3.4 illustrates an example of this function when congestion occurs in path between N_1 and N_4 . In this multicast tree, each node is requesting

number of layer in a range (ex. $L_1 = 8$ to 10). In normal state, each node receives the data with highest requested quality. In this case N_4 receives 8 layers from N_1 . When N_4 detects that path between N_1 and N_4 has congestion, N_4 drops the receiving layer the from the top, one by one to avoid the congestion. For detecting the congestion between nodes, existing method can be used [28].

Not only for avoiding congestion, by setting a range for requesting data, it could handle a various request for quality flexibly. By setting the requesting quality range narrow, it has guarantee in quality but has more possibility of switching nodes in multicast tree. By setting the requesting quality range wide, it has less chance of switch in multicast tree and offers stable service but the receiving quality may change often.

Chapter 4

Design of LOLCAST

This chapter illustrates the protocol design of LOLCAST in detail. First protocol parameters maintained by each node is discussed. Next, each of the message used in LOLCAST is illustrated in detail. Next, each step for Join Procedure and Leave Procedure is discussed. Last, design of our proposed Multi-path layer distribution method and congestion avoidance method is illustrated.

4.1 Tree Parameters

Multicast tree constructed by LOLCAST has four basic parameters. This tree is maintained by the source node, and each of the parameters are shown below.

L_{max} : Maximum Number of Layers

Maximum number of layers included in the data sent by source node. Value takes between 0 and L_{max} .

D_{max} : Maximum Tree Depth

Maximum tree depth in the multicast tree. Leaf node with the highest depth will have D_{max} .

B : Bandwidth of Layer

Bandwidth of each layer included in the data. If the data has i layers, sum of the bandwidth through layer 0 to i is $\sum_{i=0}^{i-1} B_i$. The bandwidth of the original data sent by the source is $\sum_{i=0}^{L_{max}} B_i$.

PCS_{max} : Maximum Number of Entry in PCS

Maximum number of entries included in the Parent Candidate Set sent by the source node. This value is used when generating PCS.

4.2 Node Parameters

Nodes joining to the multicast tree has two categories: source node N_0 which will be the root in the multicast tree and others, as relay node N_i ($i > 0$). In this section, parameters used in LOLCAST is discussed. First fundamental parameters for every node is illustrated. Next source and relay node specific parameters are discussed.

4.2.1 Fundamental Parameters

This section shows the fundamental parameters for every node. It is assumed that every node has allocated unique node identifier. Each parameter is referenced with such as $N_s.field$.

***id* : Node Identifier**

Unique node identifier allocated for each node. Value is set by default.

***ly* : Requesting Layer**

Number of layers the node is requesting. Value is set by default. Value for the source node is always set to L_{max} .

***bw* : Bandwidth Left**

Bandwidth left for sending data used in LOLCAST. If there is multiple network interface, the value will take the sum of it. Value is given by the user in initial process and set by default.

***state* : Node State**

This parameter shows the state of the node. There are six types of state, *INIT*, *RUNNING*, *JOINING*, *ACCEPTED*, *ESTABLISHED* and *LEAVING*. Initial value is set to *INIT*.

***parent* : Parent Node Identifier**

Node identifier of the parent node. Initial value is set to $N_0.parent = 0$, $N_i.parent = -1$.

***C* : Set of Child Nodes**

Set of the child nodes which the node maintains ($N_i.C = \{x | x \in N, x \neq N_i\}$). Initial value is set to $N_i.C = \phi$. Information included for each entry is shown below.

***id* : Node Identifier**

Node Identifier of the child node.

ly : Requesting Layer

Number of layer requested by the child node.

state : Node State

There are five types of state for child nodes. *DOWN*, *JOINING*, *ACCEPTED*, *LEAVING* and *CONGESTED*.

4.2.2 Source node specific parameters

Source node N_0 maintains set of the node state joined to the multicast tree T . Every node N_i joining to the tree can referenced as T_i . Initial value for T is a set with only the source node T_0 included, and the parameters are $T_0.id = 0$, $T_0.dp = 0$, $T_0.ly = L_{max}$, $T_0.bw = N_0.bw$, $T_0.state = ESTABLISH$. Each entry included in T carries the information below.

id : Node Identifier

Unique node identifier of the node.

dp : Depth

Depth of the node in multicast tree. Value takes between 0 and D_{max} .

ly : Number of Layer

Number of layers the node requests for parent node or it maintains. Value takes between 1 and L_{max} .

bw : Bandwidth

Bandwidth left for sending data used in LOLCAST.

state : Node State

State of the node. Value takes *ESTABLISH*, *JOINING* and *LEAVING*.

p : Node Identifier of Parent Node

Node identifier of its parent node.

For operating T , there are two functions defined in LOLCAST, *lowerLayerNodes()* and *minDepthNode()*. Process of each functions is described.

```

1: lowerLayerNodes ( $T, l$ ) { /* T=Tree information, l=# of layer */
2:   return { $x.id$  |  $x \in T, x.ly \geq l$ };
3: }
```

Figure 4.1: lowerLayerNodes()

lowerLayerNodes() is a function to obtain set of node identifiers which has number of layer greater than specified value. Process of *lowerLayerNodes()* is illustrated briefly.

- 1: *lowerLayerNodes()* takes two variables, set of node identifiers which node is included in the multicast tree T and requesting number of layers l .
- 2: Each node identifier ($x.id$) is taken out from T which node is included in T ($x \in T$), and number of layer is greater or equal than the request l ($x.ly \geq l$).

```

1: minDepthNode ( $T, P$ ) { /* T=Tree information, P=set of node id */
2:   return i s.t. Mini∈P(Ti.dp);
3: }
```

Figure 4.2: minDepthNode()

Correspondingly, *minDepthNode()* is a function to obtain a set of node identifier which has smallest depth inside certain set of node identifiers. Process of *minDepthNode()* is illustrated briefly.

- 1: *minDepthNode()* takes two variables, set of node identifiers which node is included in the multicast tree T and certain set of node identifiers P .
- 2: A node identifier of a node i which has minimum depth inside set P is taken out ($Min_{i \in P}(T_i.dp)$) and returned.

4.2.3 Relay node specific parameters

Relay node has one node specific parameter.

P : Parent Candidate Set

Set of node identifier of parent candidate nodes obtained from source node. Initial value is set to $P = \phi$.

4.2.4 Node Initialization

Every node joining the tree has a initializing phase before the protocol process starts. In this section, two functions *LinkUp()* and *LinkDown()* is discussed.


```

1: LinkUp (i) { /* i=node id */
2:   if (i = 0) {
3:      $N_0.state \leftarrow RUNNING$ ; /*  $INIT \rightarrow RUNNING$  */
4:   } else {
5:      $N_i.state \leftarrow JOINING$ ; /*  $INIT \rightarrow JOINING$  */
6:      $send\_JoinRequest(0, i, N_i.ly, N_i.bw)$ ;
7:   }
8: }

```

Figure 4.3: linkUp()

linkUp() is called when the nodes link is up. *linkUp()* sets the initial state and starts the process to join.

- 1: *linkUp()* require one variable, node identifier *i*.
- 2-3: If the node is source node, state is set to *RUNNING* from *INIT*.
- 4-6: If the node is relay node, state is set to *JOINING* from *INIT*. *sendJoinRequest()* is called with the variables required for Join Procedure.

```

1: LinkDown (i) { /* i=node id */
2:    $N_i.state \leftarrow INIT$ ; /*  $** \rightarrow INIT$  */
3:   initialize all data in  $N_i$ ;
4: }

```

Figure 4.4: linkDown()

linkDown() is called when protocol process finishes and the link got down.

- 1: *linkDown()* require one variable, node identifier *i*.
- 2: Node state is set to *INIT*.
- 3: All parameters maintained by the node is initalized.

4.3 Node State

For the multicast tree consistency state for the node is maintained in three different part as stated in Section 4.1 and 4.2. Figure 4.5 illustrates the state diagram for each node. Figure 4.6 illustrates the state diagram of nodes which maintained in the tree structure.

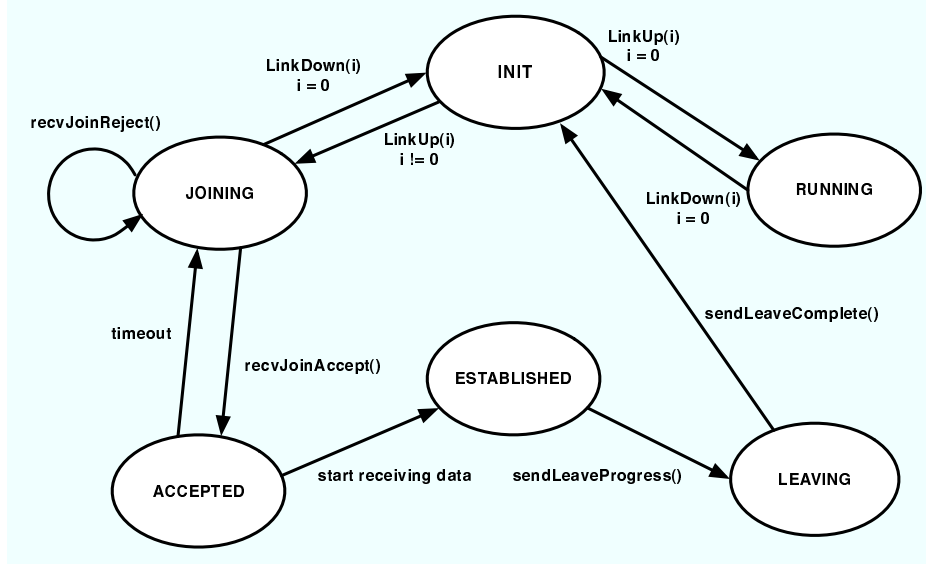


Figure 4.5: State diagram for node

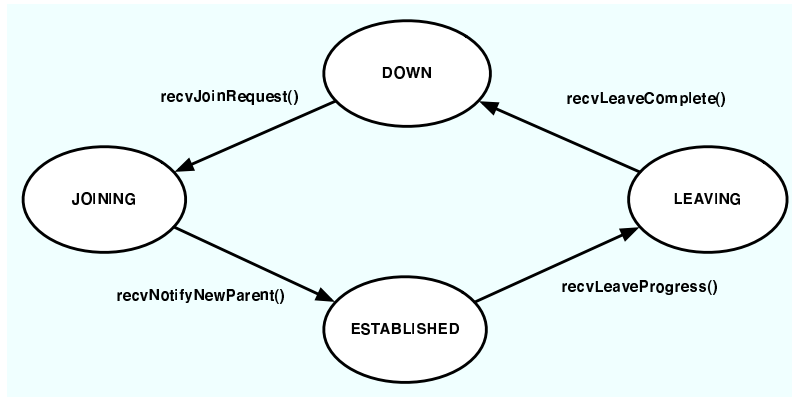


Figure 4.6: State diagram for node in tree structure

4.4 Messages

LOLCAST constructs and maintains the multicast tree by message passing between other nodes. This section represents each of the messages defined in LOLCAST.

4.4.1 Join Request Message

Join Request message is a message for a node which joining to the tree to request the source node for which parent node to join to. This message is only sent by relay node and received only by source node. In the process of Join Request, function for generating the set of candidate parent nodes, *extractPCS()* is called. First the function *extractPCS()* is discussed and next, process of *JoinRequest()* is shown.

```

1: extractPCS ( $i, l, S$ ) { /*  $i$ =request node,  $l$ =# of layers */
2:    $S \leftarrow \phi$ ;
3:    $A \leftarrow \text{lowerLayerNodes}(T, l)$ ;
4:   for ( $|A| > 0$  ()) {
5:      $j \leftarrow \text{minDepthNode}(T, A)$ ;
6:      $A \leftarrow A - \{j\}$ 
7:     if ( $(\sum_{k=0}^{l-1} B_k < T_j.bw$  and  $T_j.id \neq i$ )
8:        $S \leftarrow S \cup j$ ;
9:       if ( $|S| \geq PCS_{max}$ )
10:        return;
11:   }
12: }
```

Figure 4.7: *extractPCS()*

extractPCS() evaluates the node with four conditions taken out from T maintained by the source node. *extractPCS()* is called only by the source node. The conditions are: node has larger number of layers than the request, node has enough to send requesting data, node is not the node requesting for PCS, and the node has minimal depth inside above all. Figure 4.4.1 illustrates the process.

- 1:** *extractPCS()* takes three arguments. Node identifier of the node requesting for PCS i , requesting number of layers l and PCS S .
- 2:** Set the extracting PCS to ϕ .
- 3:** Set of node identifiers of nodes with number of layer greater than l is taken out by *lowerLayerNodes*(T, l) from T into a new set A .
- 4-6:** Until A is unavailable, node j which has the minimal depth inside A is taken out by *minDepthNode*(T, A). Next, j is cleared from A .
- 7-8:** Two conditions, $\sum_{k=0}^{l-1} B_k < T_j.bw$ and $T_j.id \neq i$ are evaluated. First is total bandwidth of the data with l layers ($\sum_{k=0}^{l-1} B_k$) is smaller than the bandwidth left for j ($T_j.bw$). Second is node identifier for j ($T_j.id$) is not equal to the node identifier of node called *extractPCS*, i . If both meets conditions, j is added to the generated PCS S .
- 9-10:** Number of entries in PCS S is evaluated. If number of entries are greater or equal to PCS_{max} , parameter in tree structure, extracted S is returned. If not, another node j is taken out from A in line 5.

```

1: recvJoinRequest ( $i, j, l, b$ ) { /*  $i, j$ =nodeid,  $l$ =# of layer,  $b$ =bandwidth */
2:   if ( $i = 0$ ) {
3:      $T_j.id \leftarrow j$ ;
4:      $T_j.dp \leftarrow 0$ ; /* initialized value */
5:      $T_j.ly \leftarrow l$ ;
6:      $T_j.bw \leftarrow b$ ;
7:      $T_j.p \leftarrow -1$ ;
8:      $T_j.C \leftarrow \phi$ ;
9:     if ( $\sum_{k=0}^{l-1} B_k < N_0.T_0.bw$ ) {
10:       $send\_JoinAccept(j, i)$ ;
11:       $T_j.state \leftarrow JOINING$ ; /*  $INIT \rightarrow JOINING$  */
12:       $N.i.C \leftarrow N.i.C \cup \{(j, l, JOINING)\}$ ; /* add new child */
13:       $N_0.bw = N_0.bw - \sum_{k=0}^{l-1} B_k$ ;
14:      /* set timeout in  $T_{out}$  for node  $j$  */
15:    } else {
16:       $PCS \leftarrow extractPCS(l)$ ;
17:      if ( $|PCS| = 0$ ) {
18:         $send\_JoinReject(j, i)$ ;
19:         $T_j.state \leftarrow INIT$ ; /*  $INIT \rightarrow INIT$  */
20:      } else {
21:         $send\_JoinRedirect(j, i, PCS)$ ;
22:         $T_j.state \leftarrow JOINING$ ; /*  $INIT \rightarrow JOINING$  */
23:        /* set timeout in  $T_{out}$  for node  $j$  */
24:      }
25:    } else {
26:      if ( $N_i.state = ESTABLISHED$  and  $\sum_{k=0}^{l-1} B_k \leq N_i.bw$  and  $N_i.ly \geq l$ )
27:      {
28:         $send\_JoinAccept(j, i)$ ;
29:         $N_i.bw = N_i.bw - \sum_{k=0}^{l-1} B_k$ ;
30:        /* set timeout in  $T_{out}$  for node  $j$  */
31:         $N.i.C \leftarrow N.i.C \cup \{(j, l, JOINING)\}$ ; /* add new child */
32:      } else {
33:         $send\_JoinReject(j, i)$ ;
34:      }
35:    }

```

Figure 4.8: JoinRequest()

- 1:** *JoinRequest()* takes four variables, node identifier of source node i , node identifier of requesting node j , requesting number of layer l , bandwidth left b .
- 2:** If the node received Join Request message is N_0 , which is the source node, goes to line 4. Else if the node is other relay nodes, goes to line 20.
- 3-8:** Parameters of T_j is set to be prepared for joining to the tree. Depth of T_j is set to 0 which is only for an initial value.
- 9-10:** Checks if the source node can be a parent. If bandwidth left for source node ($N_0.T_0.bw$) is greater than the total bandwidth of requesting data from j ($\sum_{k=0}^{l-1} B_k$), a message to acknowledge the join *sendJoinAccept*(j, i) is called and goto next line.
- 11:** State of T_j in the tree structure is set to *JOINING* from *INIT* to lock from changes.
- 12:** Child node is added to source node. In detail, information of N_j is inserted to set of child nodes of the source node $N_0.C$. Node identifier j , requesting layer l and the state *JOINING* is included.
- 13:** Bandwidth left for N_0 is reduced. Total data bandwidth calculated from requesting layer l ($\sum_{k=0}^{l-1} B_k$) is subtracted.
- 14:** Timeout timer T_{out} is started for node N_j .
- 15-16:** If condition in line 9 is not satisfied, *extractPCS*(j, l, PCS) is called to find other candidate parent nodes to join.
- 17-18:** If generated *PCS* in line 13 does not have any entry, *sendJoinReject*(j, i) is called to report that there are no nodes satisfy the request.
- 19:** State of T_j in the tree structure is switched back to *INIT*.
- 20-21:** If generated *PCS* in line 13 have more than one entry, *sendJoinRedirect*(j, i, PCS) is called to redirect the node N_j to another parent node.
- 22:** State of T_j in the tree structure is set to *JOINING* from *INIT* to lock from changes.
- 23:** Timeout timer T_{out} is started for node N_j .

- 25-27:** If the node received Join Request message is relay node, following conditions are evaluated. If state of N_i is *ESTABLISHED*, bandwidth left for N_i is larger than the requesting data and layer maintained by N_i is larger than the requesting layer l . If all meet condition, *sendJoinAccept()* is called and else goes to line 31.
- 28:** Bandwidth left for N_i is reduced. Total data bandwidth calculated from requesting layer l ($\sum_{k=0}^{l-1} B_k$) is subtracted.
- 29:** Timeout timer T_{out} is started for node N_j .
- 30:** Child node is added to node N_j . In detail, information of N_j is inserted to set of child nodes of the source node $N_i.C$. Node identifier j , requesting layer l and the state *JOINING* is included.
- 31-32:** If condition in line 26 is not satisfied, *sendJoinReject()* is called for the node to report that join request failed.

4.4.2 Join Accept Message

Join Accept message is a message for candidate parent node (including the source node) to reply Join Request message that the request is accepted. This message is sent by source node and relay node and received only by relay node.

```

1: recv_JoinAccept ( $i, j$ ) { /*  $i$ =nodeid( $i \neq 0$ ),  $j$ =parent-nodeid */
2:    $N_i.parent \leftarrow j$ ;
3:    $N_i.state \leftarrow ACCEPTED$ ; /*  $JOINING \rightarrow ACCEPTED$  */
4:   sendNotifyNewParent(0,  $i, j$ );
5: }
```

Figure 4.9: JoinAccept()

- 1:** *recvJoinAccept()* takes two variables. Node identifier of new relay node i and node identifier of parent node j .
- 2:** New node N_i , sets its parent node id to j ($n_i.parent \leftarrow j$).
- 3:** State of N_i is set to *ACCEPTED* from *JOINING*.
- 4:** *sendNotifyNewParent*(0, i, j) is called for new relay node N_i to notify the source node that join to a parent node is completed.

4.4.3 Notify New Parent Message

Notify New Parent Message is a message for new relay node to notify the source node that node is joined to a parent node. Notify Parent Message operates the tree structure and add the connection between new relay node and the parent node. This message is sent only by the relay node and received only by the source node.

```

1: recv_NotifyNewParent ( $i, j, k$ ) { /*  $i,j$ =nodeid( $i=0$ ), $k$ =parent-nodeid */
2:    $T_j.parent \leftarrow k$ ;
3:    $T_j.state \leftarrow ESTABLISHED$ ; /*  $JOINING \rightarrow ESTABLISHED$  */
4:    $sendNotifyAccepted(i, j, k)$ ;
5: }
```

Figure 4.10: NotifyNewParent()

- 1: *NotifyNewParent()* takes tree variables. Node identifier of source node i , new relay node j and new parent node k .
- 2: State of N_j is set to *ESTABLISHED* from *JOINING*.
- 3: $sendNotifyAccepted(i, j, k)$ is called to notify the new relay node that operation done to the tree finished correctly.

4.4.4 Notify Accepted Message

Notify Accepted message is a message for source node to notify the relay node that operation to the tree is finished correctly. New relay node finishes join procedure by receiving this message. This message is only sent by relay node and received by source node.

```

1: recvNotifyAccepted ( $j, k$ ) { /*  $i=0, j$ =child, $k$ =parent-nodeid */
2:    $N_j.parent \leftarrow k$ ;
3:    $sendNotifyAck(0, j)$ ;
4: }
```

Figure 4.11: NotifyAccepted()

- 1: *recvNotifyAccepted()* takes two variables. Node identifier of new relay node j and node identifier of parent node k .

- 2: Node N_j sets its parent node to k .
- 3: Node N_j sends back to source node N_0 acknowledge message $sendNotifyAck(0, j)$ is called.

4.5 Join Procedure

In this section, protocol process for Join Procedure is illustrated. First outline for the Join Procedure is shown by the message passing diagram. Next example of a new relay node joining to the tree is discussed step by step.

4.5.1 Message passing in Join Procedure

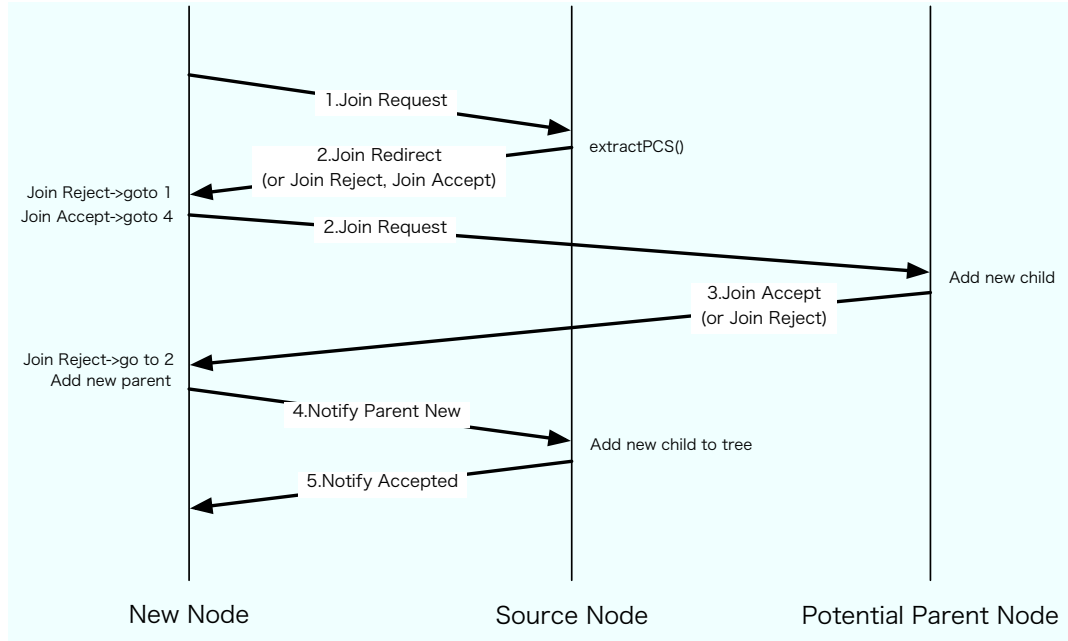


Figure 4.12: Message passing in Join Procedure

4.5.2 Example of Join Procedure

Figure 4.13, 4.14, 4.15 illustrates a case when N_4 joins to the multicast tree. Solid line stands for data path and dotted line stands for message path. This tree has two parameters, $L_0 = 10$ and $c_{min} = 2$. For simplicity, c_{max} for every node is set to 2. Join Procedure is done by messaging between nodes.

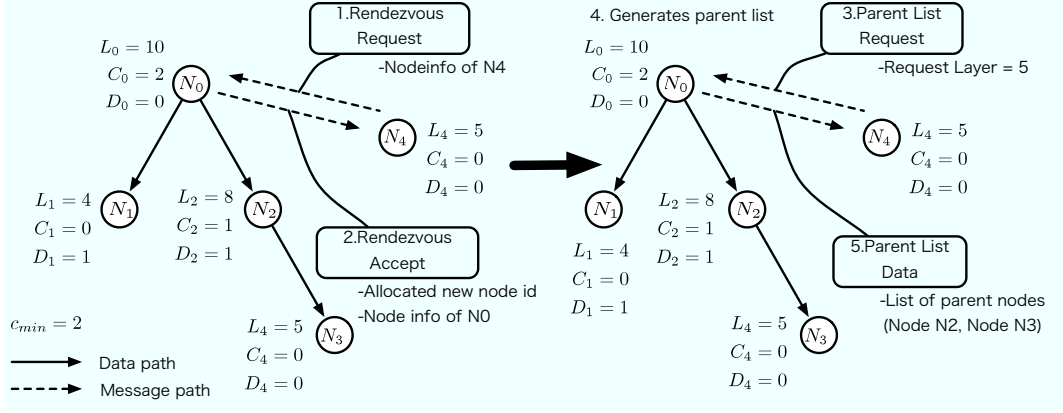


Figure 4.13: Join Procedure 1

For the first step, we assume that new relay node N_4 can acquire source node N_0 's address and max number of layer, L_0 which source could offer. N_4 sends Rendezvous Request to N_0 including its own node information. Source node generates a unique node identifier and sends back to N_4 using Rendezvous Accept message illustrated in Figure 4.13. Next, N_4 sends Parent List Request to N_0 including the requesting number of layer ($L_4 = 5$).

Parent list is a list of node which is capable to be parent node, generated by source node. Source node generates parent node list from several parameters using tree structure, which is sorted by depth. There are mainly two conditions to suffice for adding to parent node list. (a) Node has open slot in number of children ($c_{min} \leq C_i < c_{max}$). (b) Node has enough layers suffice the request of new relay node ($L_i \geq L_{new}$). (c) Node is not in the leave node list (nodes which are trying to leave from the tree are added). N_0 sends back the generated parent list including N_2 and N_3 to N_4 by Parent List Data message.

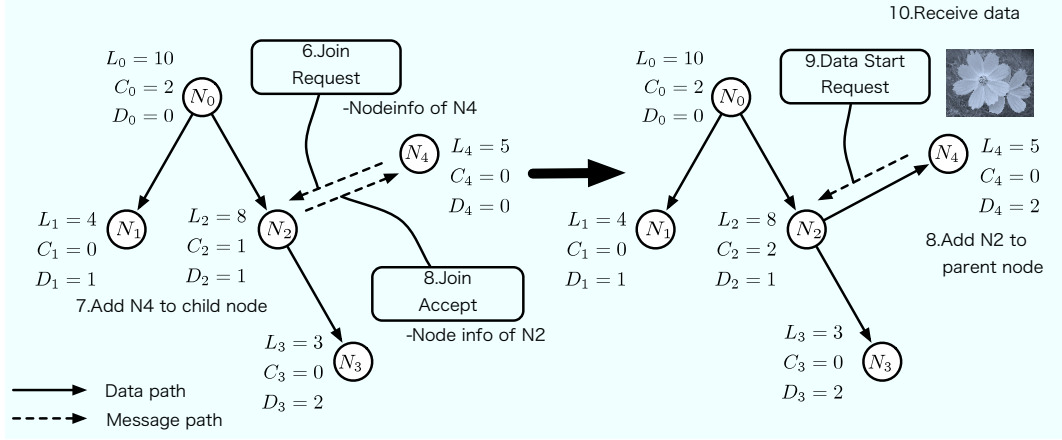


Figure 4.14: Join Procedure 2

N_4 node picks N_2 from parent list and sends Join Request including node information of N_4 as illustrated in Figure 4.14. N_2 rechecks the parameters if it can sustain the request, and becomes parent node for N_4 . N_2 adds the N_4 's node identifier, address, and other node information as children node. Next, N_2 sends Join Accept message to N_4 . N_4 adds node information of N_2 to its data structure as parent node. Finally N_4 sends Data Start Request to N_2 , and data transmission starts.

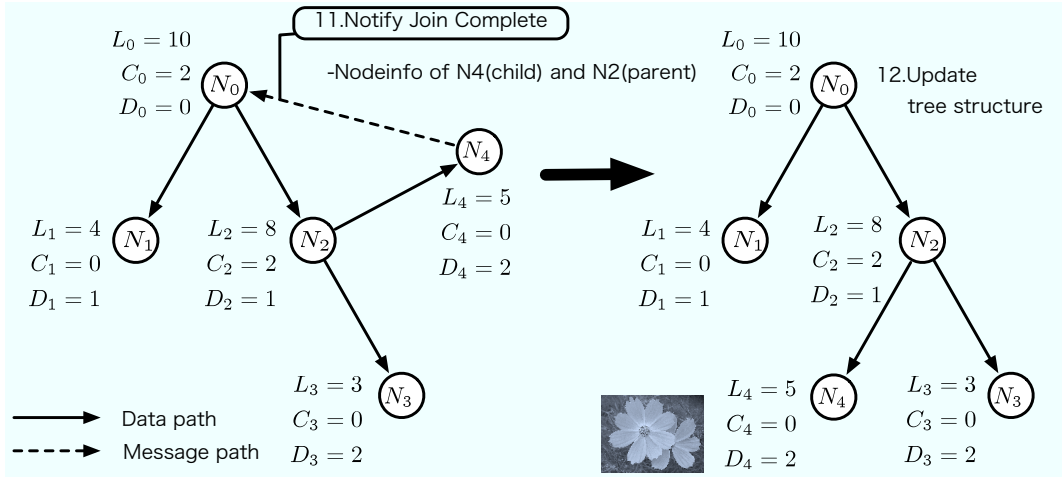


Figure 4.15: Join Procedure 3

After N_4 joined to the multicast tree, N_4 sends Notify Join Complete message to N_0 for updating the tree structure which it maintains, as illustrated in Figure 4.15.

4.6 Leave Procedure

In this section, protocol process for Leave Procedure is illustrated. First outline for the Leave Procedure is shown by the message passing diagram. Next example of a relay node leaving from the tree is discussed step by step.

4.6.1 Message passing in Leave Procedure

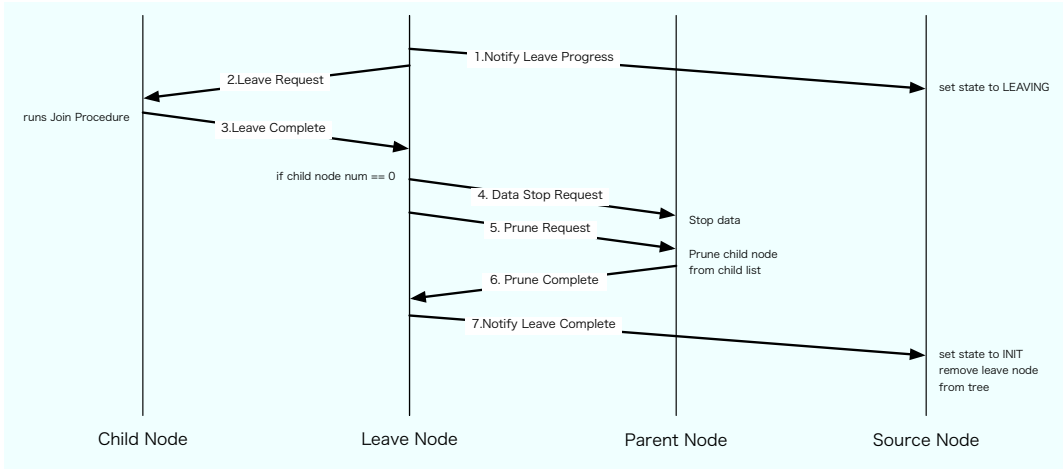


Figure 4.16: Message passing in Leave Procedure

4.6.2 Example of Leave Procedure

Figure 4.17, 4.18 illustrates a case when N_4 leaves from the multicast tree. Parameters are same as in Join Procedure. For simplicity, c_{max} for every node is set to two. In Leave Procedure, it is desired for the leaving node not to make an effect to the data streaming of other nodes.

First, leaving node N_4 sends Notify Leave Progress message to the source node N_0 as illustrated in Figure 4.17. This message notifies the source node that its state must be locked, such as not to include the node to the parent list. N_0 adds N_4 to the leave node list to lock the node from modification. Next, N_4 sends Leave Request to the child node N_5 to notify that N_4 is leaving.

When N_5 receives Leave Request, it runs Join Procedure to find other parent node. N_5 rejoins to the new parent N_1 , and start receiving the data from it. In this period, N_5 is receiving two data stream redundantly from N_4 and N_1 . After switching the data stream from N_4 to N_1 , N_5 sends Data Stop Request message to N_4 and N_4 stops sending data to N_5 . Next, N_5 sends

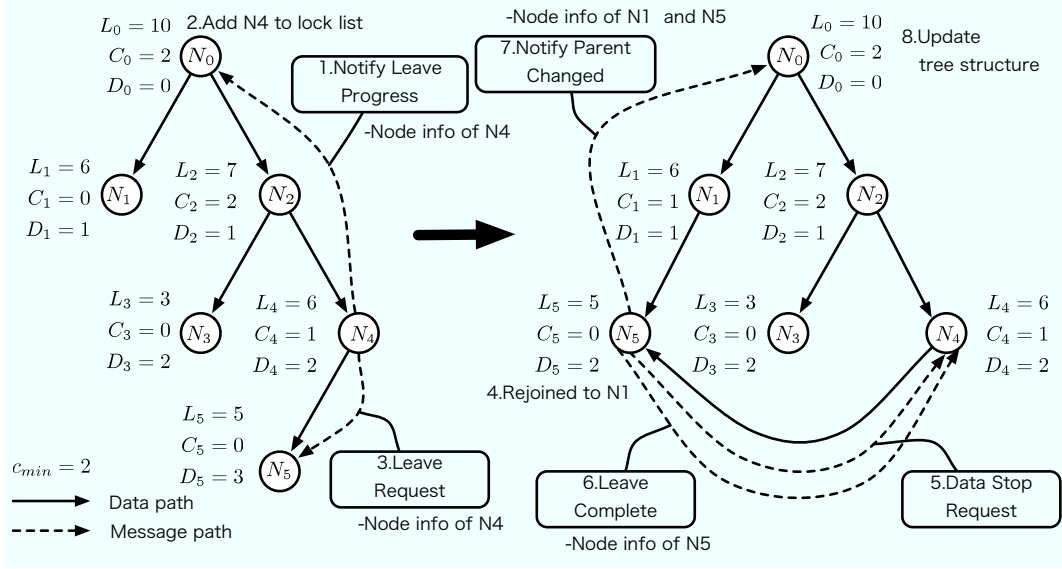


Figure 4.17: Leave Procedure 1

Leave Complete message to N_4 to notify N_4 that N_5 has found a new parent node and receiving data properly. Finally N_5 sends Notify Parent Changed message including the node information of both N_5 and N_4 to request the source node N_0 to update the multicast tree. N_0 updates the multicast tree to the current state.

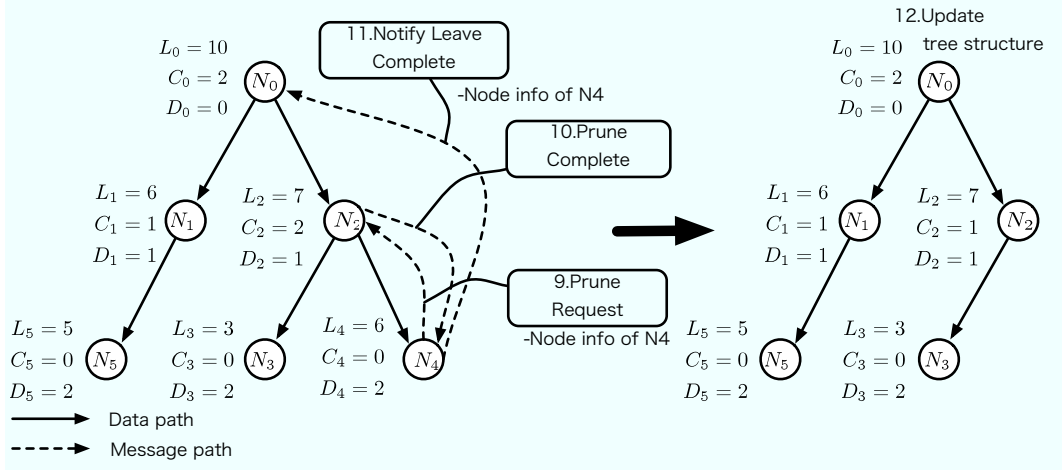


Figure 4.18: Leave Procedure 2

After N_4 confirms that it has no children nodes, N_4 sends Prune Request message to its parent node N_2 . N_2 deletes node information of N_4 from its data structure and sends back Prune Complete to N_4 . Finally N_4 sends

Notify Leave Complete to N_0 and N_0 deletes node information of N_4 from multicast tree structure.

Chapter 5

Implementation of LOLCAST

This chapter represents the implementation of streaming application using LOLCAST protocol. Implemented application serves data with multiple media-type (text data, audio data, low quality video data and high quality video data) with four layers by using abstract data structure. First environment of the implementation is illustrated. Next, system overview of the application is shown. Next, each of the functional modules to construct the system are described. Last, user interface for using the application is illustrated.

5.1 Implementation Environment

LOLCAST application is implemented as a content streaming streaming application. Table 5.1 shows the implementation environment. The operating system used in the implementation is MacOSX 10.4.8. C++ is used for the programming language, and Standard Template Library is used for implementing the data structure.

Table 5.1: Environment for implementation

CPU	Intel Core Duo 1.66Ghz
Memory	2GB
OS	MacOSX 10.4.8
Language	C++
Library	C++ Standard Template Library
Compiler	4.0.0 20041026 (Apple Computer, Inc. build 4061)

5.2 Implementation Overview

In this section, overview of the LOLCAST application is illustrated. LOLCAST application is consist of four modules. Protocol Processing Module, Application Module, Network Module and Simulator Module. Figure 5.1 illustrates the relationship between each modules. Each of the modules are used to run the application in two modes: Application Mode and Simulator Mode. Application Mode runs the actual content streaming application by sending messages to other nodes. Simulator Mode runs as stand alone application to generate virtual nodes inside, and messaging between them to simulate the protocol. In Application Mode, Protocol Processing Module, Application Module and Network Module are used to run. In Simulator Mode, only the Protocol Processing Module and Simulator Module are used. Functions of each of the module is illustrated below.

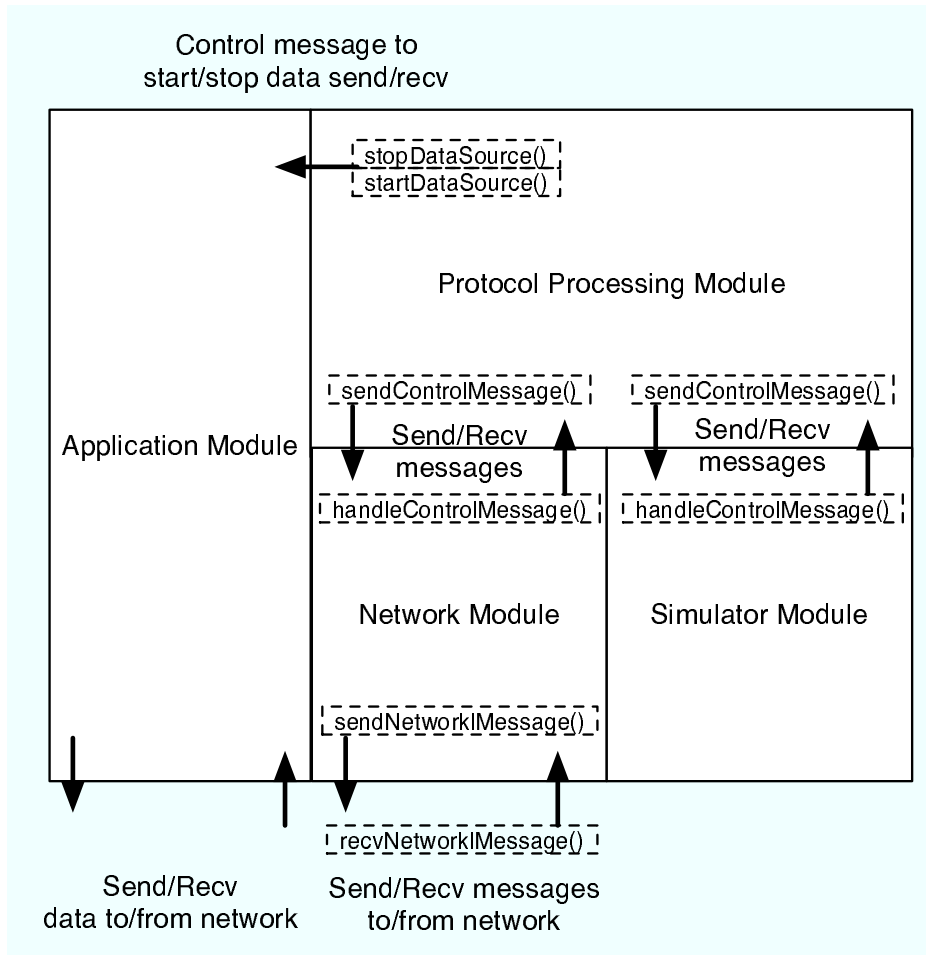


Figure 5.1: LOLCAST Application Modules

- **Protocol Processing Module**

Main function of Protocol Processing Module is to run the protocol process of LOLCAST to maintain control topology. Protocol processes are triggered by the received messages from Network Module or Simulator Module. It operates the data structure in LOLCAST and sends back messages if necessary. In addition, it controls the Application Module to send and receive content data.

- **Application Module**

Application Module maintains the data topology in LOLCAST application. Application modules send or receive the content data by the request from Protocol Processing Module. In addition, Application Module has a function to view the content in appropriate method for certain media type.

- **Network Module**

Network Module maintains connection between nodes to maintain the control topology and send or receive message packets from network. Messages received from the Protocol Processing Module are packetized and send to the destination node. Messages received from the network are taken out from the packet and given to the Protocol Processing Module.

- **Simulator Module**

Simulator Module generates virtual nodes to simulate the protocol process. Protocol process is done by messaging between virtual nodes communicating with Protocol Processing Module.

5.3 Protocol Processing Module

Protocol Processing Module maintains the tree structure and control topology in LOLCAST. This tree structure and control topology is constructed and maintained by messaging between nodes. Protocol Processing Module is the core module which runs the actual LOLCAST protocol process to establish a service infrastructure.

In this section, implementation of Protocol Processing Module is illustrated. First, data structure maintained by both source and relay node is described. Next, message format and methods which operates the data structure are illustrated. After that, methods to construct the multicast tree by handling the tree structure is illustrated particularly.

5.3.1 Data Structure

This section illustrates the implementation of data structure maintained by nodes. First elemental structure used in both source and relay node is described. Next, data structure maintained by each node are illustrated.

```

struct nodeInfo {
    nodeid_t    nodeId;
    address_t   address;
    int        layer;
};

typedef list<nodeInfo *> nodeInfoSeq;

```

Figure 5.2: nodeInfo Structure

Figure 5.2 represents structure of **nodeInfo**. **nodeInfo** is used to maintain child nodes and parent node information by both source and relay node. **nodeInfo** includes, node identifier, network address and requesting layer of the node. In addition, list of **nodeInfo** is defined as **nodeInfoSeq**.

```

struct treeNodeInfo{
    nodeid_t    nodeId;
    int        nodeState;
    address_t   address;
    int        layer;
    bandwidth_t bandwidthLeft;
    int        depth;
    LCNode::treeNodeInfoSeq  nodeInfoChild;
};

typedef map<nodeid_t, treeNodeInfo *> treeNodeInfoMap;
typedef list<treeNodeInfo *> treeNodeInfoSeq;
typedef list<treeNodeInfo> treeNodeInfoInstanceSeq;

```

Figure 5.3: treeNodeInfo Structure

Figure 5.3 represents structure of **treeNodeInfo**. **treeNodeInfo** is used to maintain the multicast tree structure and only used by the source node. **treeNodeInfo** includes fundamental variables to maintain the tree. The data included are, unique node identifier, state of the node, network address, requesting layer, bandwidth left and depth inside the tree. These parameters are used to select the optimal parent node inside the tree. In addition, list of **treeNodeInfo**, which maintains the information for its child node is included as **nodeInfoChild**. Map of node identifier and **treeNodeInfo** is defined as **treeNodeInfoMap** and list of **treeNodeInfo** is defined as **treeNodeInfoSeq**.

```

struct layerInfo {
    int           dataType;
    bandwidth_t  dataBandwidth;
    double       dataSendRate;
    char         dataName[100];
};

typedef list<layerInfo *> layerInfoSeq;

```

Figure 5.4: layerInfo Structure

Figure 5.4 represents structure of **layerInfo**. **layerInfo** is used to maintain information of each layer in the content first maintained by the source node and reported to other relay nodes afterward. First entry of **layerInfo** includes the type of the data included in the layer, for example, text data, audio data, video data and layered coded video data. In addition, bandwidth used for the layer, and name of the layer is included. List of **layerInfo** is defined as **layerInfoSeq**.

Source Node

```

class LCNode {
    private:
        /* node information */
        int           nodeType = LCNODE.SOURCE;
        int           nodeState;
        nodeid_t      nodeId = NODEID.SOURCE;
        address_t      address;
        bandwidth_t    bandwidthSend;
        bandwidth_t    bandwidthLeft;
        int           layer;
        nodeInfoSeq    nodeInfoChild; // child node information
        layerInfoSeq    dataInfo; // data information

        /* source specific information */
        double         rateOutbound; // rate for sending data
        nodeid_t        nodeIdCurrent; // current node identifier
        treeNodeInfoMap treeInfo; // multicast tree structure

        /* abbr */
        .

};

```

Figure 5.5: Data structure maintained by source node

Figure 5.5 represents data maintained by the source node. As illustrated in Section 4.2.2, source node maintains the multicast tree structure. This tree is implemented as a map of **treeNodeInfo**, which is **treeInfo**. Information for the streaming content is implemented as a list of **layerInfo**, which is

dataInfo. Parameter **rateOutbound** is added for making a constraint in receiving number of layers. This parameter is for not having a node only receive and not relaying the data, which will cause the multicast tree not join-able. **rateOutbound** takes value greater than 1.0, which multiplied by the receiving data bandwidth to show total data bandwidth to send. In addition parameter **nodeIdCurrent** saves the current unique node identifier.

Relay Node

```

class LNode {
private:
    /* node information */
    int         nodeType = LCNODE_RELAY;
    int         nodeState;
    nodeid_t    nodeId;
    address_t    address;
    bandwidth_t bandwidthSend;
    bandwidth_t bandwidthLeft;
    int         layer;
    nodeInfoSeq nodeInfoChild; // child node information
    layerInfoSeq dataInfo; // data information

    /* relay specific information */
    bandwidth_t bandwidthRecv;
    double      rateBackup; // backup rate
    nodeInfo    *nodeInfoSource; // source node information
    nodeInfo    *nodeInfoParent; // parent node information
    nodeInfoSeq *nodeInfoPotentialParent; // parent candidate list

    /* abbr */
    .
};

```

Figure 5.6: Data structure maintained by relay node

Figure 5.6 represents data maintained by the relay node. As illustrated in Section 4.2.3, relay node maintains list of parent candidate list received from the source node in Join Request. Parent candidate list is implemented as list of **nodeInfo**. Parameter **rateBackup** is added to declare the percentage of receiving bandwidth used for receiving the backup data. This parameter takes value from 0.1 to 0.9. Relay node also maintains node information of source node and parent node by **nodeInfo**.

5.3.2 Message Format and Message Handling Methods

In this section, messages format and methods used in LOLCAST application is illustrated briefly.

Message Format

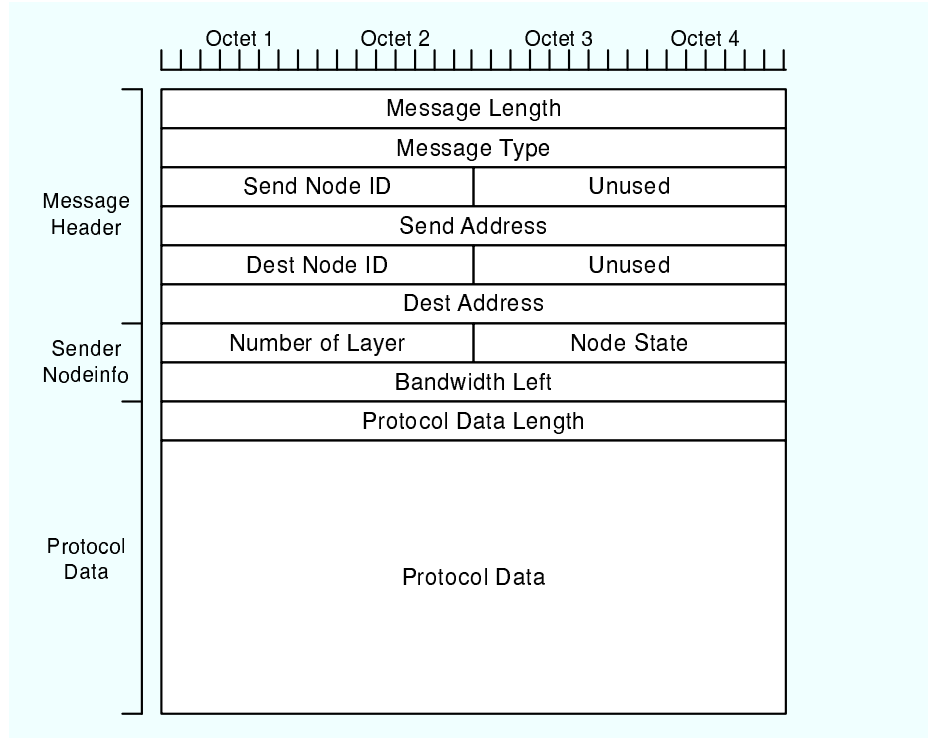


Figure 5.7: Message Format

Message format used in LOLCAST is generally divided into three parts as illustrated in Figure 5.7. Message Header, Sender Nodeinfo and Protocol Data. Message Header includes the fundamental information to send the message. Node identifier and address of both sender and destination node is included. Sender nodeinfo includes the relay nodes information for joining to the multicast tree. These fields are used for source node to find appropriate parent node for it. Protocol data includes other data such as information of streaming data and parent candidate list.

Message Methods

Figure 5.8 represents the methods called when receive or send messages. Send and receive method are defined for each message. All messages received from Network Module or Simulator Module is given to *handleControlMessage()*. *handleControlMessage()* switches the message by its message type to each of the message receive method. In case of sending a message, *sendControlMessage()* is called to pass the message to Network Module or Simulator Module.

```

/* message top handle functions */
void handleControlMessage(LCMessage *lcmg);
void sendControlMessage(LCMessage *lcmg);

/* message handling functions */
void setControlMessageHeader(LCMessage *lcmg, address_t destaddr);
void recvTreeinfoRequest(LCMessage *lcmg);
void recvTreeinfoReply(LCMessage *lcmg);
void recvJoinRequest(LCMessage *lcmg);
void recvJoinAccept(LCMessage *lcmg);
void recvJoinReject(LCMessage *lcmg);
void recvJoinRedirect(LCMessage *lcmg);
void recvNotifyParentNew(LCMessage *lcmg);
void recvNotifyParentAccept(LCMessage *lcmg);
void recvNotifyParentReject(LCMessage *lcmg);
void recvNotifyStateUpkeep(LCMessage *lcmg);
void recvNotifyStateAccept(LCMessage *lcmg);
void recvNotifyStateReject(LCMessage *lcmg);
void recvDataRequest(LCMessage *lcmg);
void sendTreeinfoRequest(void);
void sendTreeinfoReply(address_t addr);
void sendJoinRequest(address_t addr);
void sendJoinAccept(address_t addr, nodeid_t id);
void sendJoinReject(address_t addr);
void sendJoinRedirect(address_t addr, nodeInfoSeq *pcs, nodeid_t id);
void sendNotifyParentNew(address_t addr);
void sendNotifyParentAccept(address_t addr);
void sendNotifyParentReject(address_t addr);
void sendNotifyStateUpkeep(void);
void sendNotifyStateAccept(address_t addr);
void sendNotifyStateReject(address_t addr);
void sendDataRequest(address_t addr);

```

Figure 5.8: Message Handling Methods

For example in case of receiving Join Request message in Join Procedure, first *handleControlMessage()* is called. *handleControlMessage()* passes the message to correct receive method which is *recvJoinRequest()*. After running the protocol process triggered by the message, *sendJoinAccept()* is called and given to *sendControlMessage()*.

For implementing a application using LOLCAST protocol, four implementation specific messages are added. Treeinfo Request, Treeinfo Reply, Data Request and Data Stop Request. Treeinfo Request is a message for new relay node to request the source node for service information, which is the information of the streaming data and outbound rate. This information is send by Treeinfo Reply message. After a new relay node finished joining to the multicast tree, Data Request message is send to its parent node to request the data. As soon as parent node receives this message, parent node starts sending the data with requested number of layer. Data Stop Request message is called when relay node requests its parent to stop the stream.

5.3.3 Tree Handling Methods

In this section, tree handling methods implemented is illustrated. Tree handling methods operates the tree structure to manipulate the control topology, which is called only by the source node. First some of small utility methods are illustrated. Next, search method to find appropriate parent node in the multicast tree: *generatePCS()* and *generatePCSRejoin()* is illustrated. As above, core part which will dictate the quality of the multicast tree are done in these methods.

```
/* tree handling functions */
nodeid_t generateNodeId(void);
void setState(int state);
void generatePCS(nodeInfoSeq* pcs, nodeid_t id);
void generatePCSRejoin(nodeInfoSeq* pcs, nodeid_t id);
void addSubTree(treeNodeInfo* p, nodeid_t id, int cly,
               bandwidth_t bw, treeNodeInfoInstanceSeq *tmp);
void cutNode(treeNodeInfo *p, nodeid_t id);
void setDepth(treeNodeInfo *p, int depth);
```

Figure 5.9: Tree Handling Methods

Figure 5.9 represents the methods called when modification to the multicast tree is needed. *cutNode()* is called when Leave Procedure is completed to cut off the leaving node from the multicast tree. *setDepth()* is called when recalculation for depth is required in case of parent node switch. *setDepth()* runs inside the tree structure recursively from the given node to set new depth for subtree under it.

Parent Node Selection

In LOLCAST, source node maintains the tree structure including all nodes joining to the multicast group. Parent node selection done by the source node consequently determines the quality of the multicast tree. This section illustrates how the two core methods for selecting parent nodes: *generatePCS()* and *generatePCSRejoin()* are implemented.

generatePCS()

generatePCS() is called when source node received Join Request message from the relay node. Process of *generatePCS()* is illustrated in Figure 5.10. *generatePCS()* takes two arguments: generating parent candidate list and the joining nodes identifier.

1-6: Node information for the joining node is searched from the tree structure **treeInfo** and requesting layer and total bandwidth for the data is calculated.

12-28: Each **treeNodeInfo** entry in the tree structure are searched until temporary PCS size is greater than the defined **PCS_SIZE**. For each entry, following four condition are evaluated.

1. Node has enough layer for the request
2. Node has enough bandwidth for sending requesting layer
3. Node is not the requesting node
4. State of the node is STABLE

This loop is executed from the node which has equal layer for request to the node which has higher layer. It is necessary to search from the node which has equal layer for the request for not to burden the bandwidth for the node requesting higher layer.

31: The generated temporary PCS is sorted by the *sort()* method using the overloaded operator in line 44-50. Each node is compared by three conditions. Priority is higher from the node which has smaller layer, to the node which has smaller depth, and to the node which has larger bandwidth.

34-41: Finally the sorted PCS is inserted to the generated PCS.


```

1 void
2 LNode::generatePCS(nodeInfoSeq* pcs, nodeid_t id){
3     /* find node info and get requesting layer & data bandwidth */
4     treeNodeInfoMap::iterator itr = treeInfo.find(id);
5     int ly = itr->second->layer;
6     bandwidth_t bw = getDataBandwidth(ly);
7
8     treeNodeInfoInstanceSeq tmp;
9     int cly; // current layer
10
11     /* generate temporary PCS */
12     for(cly = ly; (unsigned int)cly <= dataInfo.size() &&
13         tmp.size() <= PCS.SIZE; cly++){
14         // add to pcs from node with equal layer to up
15         itr = treeInfo.begin();
16         for(; itr != treeInfo.end(); ++itr){
17             // add to pcs if...
18             // 1. has enough layer requesting
19             // 2. has enough bandwidth for sending requesting layer
20             // 3. node is not the sender
21             // 4. state is stable
22             if(itr->second->layer == cly && itr->second->bandwidthLeft >= bw &&
23                 itr->second->nodeId != id && itr->second->nodeState == STATE_STABLE){
24                 treeNodeInfo a = *(itr->second);
25                 tmp.insert(tmp.end(), a);
26             }
27         }
28     }
29
30     /* sort tmporary PCS by layer, depth and bandwidth */
31     tmp.sort();
32
33     /* insert sorted PCS */
34     treeNodeInfoInstanceSeq::iterator sitr = tmp.begin();
35     for(; sitr != tmp.end() && pcs->size() <= PCS.SIZE; ++sitr){
36         nodeInfo *p = New(nodeInfo);
37         p->nodeId = (*sitr).nodeId;
38         p->address = (*sitr).address;
39         p->layer = (*sitr).layer;
40         pcs->insert(pcs->end(), p);
41     }
42 }
43
44 bool operator< (const treeNodeInfo &lhs, const treeNodeInfo &rhs)
45 {
46     return (lhs.layer < rhs.layer)
47         || (lhs.layer == rhs.layer && lhs.depth < rhs.depth)
48         || (lhs.layer == rhs.layer && lhs.depth == rhs.depth
49             && lhs.bandwidthLeft > rhs.bandwidthLeft);
50 }

```

Figure 5.10: generatePCS()

generatePCSRejoin()

generatePCSRejoin() is called in the Leave Procedure when child node of a leaving node received Leave Request message. Child node asks the source node which node to rejoin to recover to the multicast tree. Source node calls *generatePCSRejoin()* when received it request. Process of *generatePCSRejoin()* is illustrated in Figure 5.11. *generatePCSRejoin()* takes two arguments: generating parent candidate list and the joining nodes identifier. Basic process in *generatePCSRejoin()* is similar to *generatePCS()*. The difference is that requesting nodes underlying subtree is ignored from the parent candidate list to prevent tree to loop.

- 1-6:** Node information for the joining node is searched from the tree structure **treeInfo** and requesting layer and total bandwidth for the data is calculated.
- 12-16:** Each **treeNodeInfo** entry in the tree structure are searched from the source node until temporary PCS size is greater than the defined **PCS_SIZE**. *generatePCSRejoin()* calls recursive function *addSubTree()* to search inside the tree defined line 32-55. For each entry, same conditions as *generatePCS()* are evaluated for adding to PCS. In case the entry equals the requesting node, entire underlying subtree is skipped from searching parent candidate node. As same as *generatePCS()* loop is executed from the node which has equal layer for request to the node which has higher layer.
- 19:** The generated temporary PCS is sorted by the *sort()* method using the overloaded operator same as in *generatePCS()*.
- 22-29:** Finally the sorted PCS is inserted to the generated PCS.

```

1  void
2  LCNode::generatePCSRejoin(nodeInfoSeq* pcs, nodeid_t id) {
3      /* find node info and get requesting layer & data bandwidth */
4      treeNodeInfoMap::iterator itr = treeInfo.find(id);
5      int ly = itr->second->layer;
6      bandwidth_t bw = getDataBandwidth(ly);
7
8      treeNodeInfoInstanceSeq tmp;
9      int cly; // current layer
10
11     /* generate temporary PCS */
12     for(cly = ly; (unsigned int)cly <= dataInfo.size() &&
13         tmp.size() <= PCS_SIZE; cly++){
14         itr = treeInfo.find(NODEID_SOURCE);
15         addSubTree(itr->second, id, cly, bw, &tmp);
16     }
17
18     /* sort tmporary PCS by layer, depth and bandwidth */
19     tmp.sort();
20
21     /* generate temporary PCS */
22     treeNodeInfoInstanceSeq::iterator sitr = tmp.begin();
23     for(; sitr != tmp.end() && pcs->size() <= PCS_SIZE; ++sitr){
24         nodeInfo *p = New(nodeInfo);
25         p->nodeId = (*sitr).nodeId;
26         p->address = (*sitr).address;
27         p->layer = (*sitr).layer;
28         pcs->insert(pcs->end(), p);
29     }
30 }
31
32 void
33 LCNode::addSubTree(treeNodeInfo *p, nodeid_t id, int cly,
34     bandwidth_t bw, treeNodeInfoInstanceSeq *tmp) {
35     // add to pcs if...
36     // 1. has enough layer requesting
37     // 2. has enough bandwidth for sending requesting layer
38     // 3. node is not the sender
39     // 4. state is stable
40     if(p->layer == cly && p->bandwidthLeft >= bw &&
41         p->nodeId != id && p->nodeState == STATE_STABLE){
42         treeNodeInfo a = *p;
43         tmp->insert(tmp->end(), a);
44     }
45     LCNode::treeNodeInfoSeq::iterator itr = p->nodeInfoChild.begin();
46     for(; itr != p->nodeInfoChild.end(); ++itr){
47         // if nodeId equals requesting node, skip its underlying subtree
48         if((*itr)->nodeId == id){
49             return;
50         } else {
51             // run recursively
52             addSubTree((*itr), id, cly, bw, tmp);
53         }
54     }
55 }

```

Figure 5.11: generatePCSRejoin()

5.4 Application Module

In this section, implementation of Application Module is illustrated. Application Module is implemented as a interface to Video Lan Client [?] which used for the viewer and constructing the data topology. Application Module communicates with VLC by telnet interface from the request received from the Protocol Processing Module. There are four functions implemented in to run the request: *sendDataSource()*, *stopDataSource()*, *startDataRelay()* and *stopDataRelay()*.

```
% new layer1 broadcast enabled
% setup layer1 input file:///home/koh39/layer1.mpg
% setup layer1 output #duplicate{dst=standard
    {access=http,mux=ts,dst=192.168.0.1:9002,name=LAYER1}}
% setup layer1 loop
% control layer1 play
```

Figure 5.12: Sample telnet commands in source node

Figure 5.12 represents sample telnet commands for source node used in Application Module. In first line new broadcast channel "layer1" is created. Next input file for channel "layer1" is set. For output HTTP streaming channel is opened with port 9002. Last, channel "layer1" starts to play and start servicing the stream.

```
% new layer1 broadcast enabled
% setup layer1 input http://192.168.0.1:9002
% setup layer1 output #duplicate{dst=display,dst=standard
    {access=http,mux=ts,dst=192.168.0.2:9002,name=LAYER1}}
% setup layer1 loop
% control layer1 play
```

Figure 5.13: Sample telnet commands in relay node

Figure 5.13 represents sample telnet commands for relay node. In this case relay node run commands to receive the stream channel "layer1", created in Figure 5.12 from source node and relay it to other nodes. For the channel input, address of the source node and port is set. Next, for the channel output not only start servicing the channel "layer1", it views the stream by setting

"dst=display". Last, channel "layer1" starts to play and also servicing the stream.

As described above, each of the layer in LOLCAST is implemented as a channel in VLC. Application Module creates and maintains multiple channel to send layered data to nodes.

5.5 Network Module

In this section, implementation of Network Module is illustrated. Network Module has four major functions: *sendNetworkMessage()*, *recvNetworkMessage()*, *serializeLCMessage()* and *deserializeLCMessage()*. *sendNetworkMessage()* and *recvNetworkMessage()* are used to send or receive the message to the network given from the Protocol Processing Module. *serializeLCMessage()* and *deserializeLCMessage()* is used to packetize or unpacketize the protocol message. For example, when a node received a message from network, data is given to *recvNetworkMessage()*. *recvNetworkMessage()* calls *deserializeLCMessage()* to convert packetized message to protocol message. Retrieved message is handed to *handleControlMessage()* inside Protocol Processing Module to run the protocol process.

5.6 Simulation Module

Simulation Module works with the Protocol Processing Module to simulate the protocol process. Virtual nodes are created in Simulation Module with certain parameters. Simulation Module communicates with Protocol Processing Module by *handleControlMessage()* and *sendControlMessage()*. Messages handed from Protocol Processing Module by *sendControlMessage()* is switched to a virtual destination node. Virtual node runs the protocol process and sends back message by *handleControlMessage()* to other virtual nodes.

5.7 User Interface

This section illustrates the user interface of LOLCAST application. LOLCAST application is implemented as a command line application. LOLCAST application takes two modes: Application Mode and Simulation Mode. User interface in each mode is described briefly.

5.7.1 Application Mode

Figure 5.7.1 illustrates the user interface for lolcast_app in Application Mode. In Application Mode, lolcast_app has source mode and relay mode to run. In source mode, it takes two arguments. Outbound bandwidth and outbound rate for the multicast tree. By executing in source mode, application waits for the Join Request message from new relay node.

```
% ./lolcast_app
Usage: lolcast_app [option]
  Server example: lolcast_app -s -o 100 -r 1.5
  Client example: lolcast_app -j serveraddress -o 100 -i 100 -b 0.2
  -s:          server mode
  -i #:        inbound bandwidth
  -o #:        outbound bandwidth
  -b [0.1-0.9]: backup rate
  -r [1 <]:    outbound rate
  -j [address]: source address
  -a [address]: self address
```

Figure 5.14: Running lolcast_app in Application Mode

In relay mode, it takes four arguments. Source nodes address, outbound bandwidth, inbound bandwidth and backup rate. By executing in relay mode, it start join to the source node by sending Join Request message to the source node address.

When receiving Treeinfo Reply message from source node, prompt illustrated in Figure 5.15 will be shown. By the data information retrieved from source node, maximum number of layer which can be retrieved is determined by its receiving bandwidth. Prompt request user to input how many layers to request with showing the information of the stream.

5.7.2 Simulator Mode

Figure 5.7.1 illustrates the user interface for lolcast_app in Simulation Mode. In Simulation Mode, it takes one fundamental argument and two optional arguments to run. Number of nodes joins to the tree is fundamental for running in simulator mode. Other arguments are optional to set the parameter. "-r" option sets the request for number of layer to random. "-l" option sets to run the Leave Procedure in random order.

```
      .
      .
dataInfoMsg.size: 4
layerInfoMsg.0: dataType= 3, dataBandwidth= 10
layerInfoMsg.1: dataType= 3, dataBandwidth= 10
layerInfoMsg.2: dataType= 3, dataBandwidth= 10
layerInfoMsg.3: dataType= 3, dataBandwidth= 10
potentialParentMsg.size: 0
-----
--INPUT LAYERNUM... MAXNUM IS 4 : ___
```

Figure 5.15: Prompt for asking number of layers to request

```
%./lolcast_app
Usage: lolcast_app [option]
  Example: lolcast_app -n 100 -s 5 -l -r
    -n: number of nodes
    -s: number of layers
    -r: random layers
    -l: random leave
```

Figure 5.16: Running lolcast_app in Simulation Mode

Chapter 6

Evaluation

This chapter illustrates the evaluation on proposed Overlay Multicast protocol LOLCAST. LOLCAST is evaluated in both qualitative and quantitative approach. For qualitative approach, verification of the protocol process and functional comparison with other proposed Overlay Multicast protocols are done. For quantitative approach, performance evaluation for the protocol process is done.

The objective of the evaluation is to prove that LOLCAST satisfied all three fundamental propositions stated in Section 1.3. First proposition “User could send and receive contents with ordinary resource environment” is evaluated by verification of the protocol process and performance evaluation for protocol process. Second proposition “User could freely select media type or quality of the content on demand” is evaluated by verification of protocol process and functional comparison. Third proposition, “Seamless content delivery” is evaluated by functional comparison.

First verification of LOLCAST protocol process is done by an experiment using the implemented LOLCAST application described in Chapter 5. Next functional comparison of the protocol with other proposed Overlay Multicast protocols is done using the fundamental propositions as the metrics. Next, performance of protocol process is evaluated by measuring the performance of Join Procedure and Leave Procedure. Last, result of the evaluation is illustrated.

6.1 Verification of Protocol Process

This section represents the verification of protocol process in LOLCAST. Verification of the protocol process is done by an experiment using LOLCAST application. This experiment assumes a case of streaming a live dance party

event. First setup for the experiment is illustrated. This includes the network configuration, data served by the source and parameters set to each node. Next, experiment is done and the results are shown.

6.1.1 Experimental Setup

This section illustrates the setup for the experiment. First network configuration for the experiment is illustrated. Next, hardware and software specification of nodes joining to the multicast tree is shown. Next, abstract layered data provided from the source node is described briefly. Last, parameters used in LOLCAST for each node is illustrated.

Network Configuration

Figure 6.1 illustrates the network configuration for the experiment. There are five relay nodes joining to the tree. Relay node A, B and C are connected to the wired network. In comparison, relay node D and E are connected to wireless network.

Hardware and Software Specification

Table 6.1: Environment for experiment

Node	CPU	Memory	OS
Source	Intel Core Duo 1.66Ghz	2GB	MacOSX 10.4.8
Relay A	PowerPC 1.42Ghz	1GB	MacOSX 10.4.8
Relay B	Intel Core Duo 1.66Ghz	2GB	MacOSX 10.4.8
Relay C	PowerPC 1.42Ghz	4GB	MacOSX 10.4.8
Relay D	PowerPC 1.67Ghz	512MB	MacOSX 10.4.8
Relay E	PowerPC 1.67Ghz	1GB	MacOSX 10.4.8

Hardware and software specification for the nodes are illustrated in Table 6.1. Relay node D and E are laptop computers and connected to the network by Wi-Fi.

Provided Data

Abstracted layered data used in this experiment consists of multiple content format data described in Section 3.1. Figure 6.2 shows the actual data format. This experiment assumes a case of streaming a live dance party event.

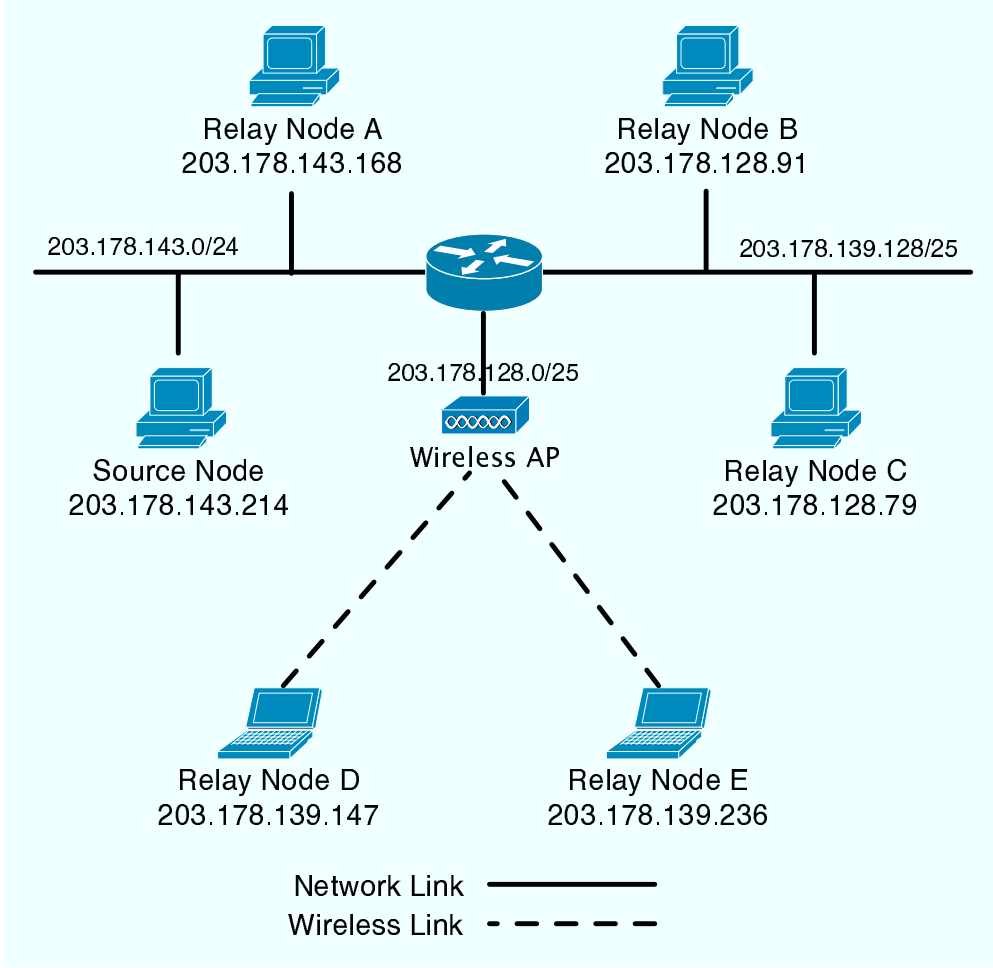


Figure 6.1: Network Topology of the experiment

Data served by the source node consists of four layers with different content format. Description of each layer is described next. The base layer offers text data supposed to be used as sending the information of the event, who is acting and where. Bandwidth consumed for this layer is set to 1. First enhancement layer offers audio data of the event and bandwidth is set to 5. Second enhancement layer offers low quality video data with bandwidth of 10. Third enhancement layer offers high quality video data with bandwidth of 25. Example of selecting four layers is shown in Figure 6.2. Base layer, first enhancement layer and third enhancement layer are played at once. Second enhancement layer is not played in this case because of both second and third enhancement layer offers same content format.

Each user will select how many layers to receive from their resource environment and their interest. User using mobile devices is assumed to select

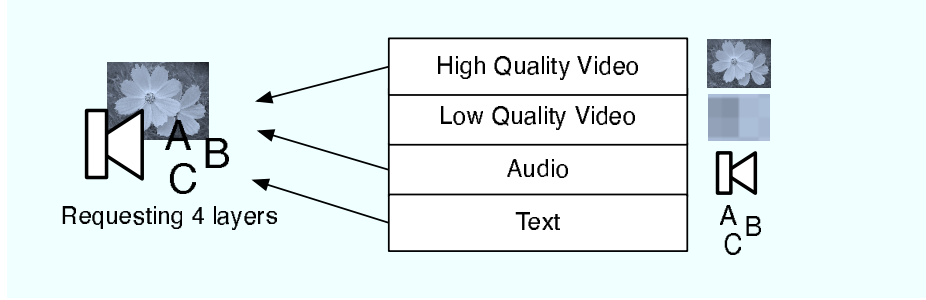


Figure 6.2: Experimental Data

one or two layers for the service. In comparison user using devices such as desktop computers is assumed to select three or full layers for the service.

Node Parameters

Table 6.2: Parameters set for experiment

Node	Outbound BW	Inbound BW	Layer	Outbound Rate	Backup Rate
Source	50	N/A	N/A	1.5	N/A
Relay A	100	100	4	N/A	0.2
Relay B	100	100	4	N/A	0.2
Relay C	100	100	3	N/A	0.2
Relay D	10	10	2	N/A	0.2
Relay E	10	10	1	N/A	0.2

Parameters for each nodes are set differently to show the disparity in resource environment. Table 6.2 illustrates parameters set for each node. In this experiment source node is assumed that does not have plenty of network resource to serve streaming service such as to hundreds of nodes, which is our targeted ordinary user. For relay nodes, we classified it into two types. Relay node A, B and C are assumed to be a node which has a large resource environment such as wired desktop computers. Relay node D and E are assumed to be a node with few resource environment such as mobile devices.

6.1.2 Experiment Result

This section illustrates the result for experiment. Experiment using LOLCAST application is done by following procedure. First source node starts the LOLCAST application in source mode and begins the service. Next each

relay node starts Join Procedure to the source node. The Join Procedure is done in the following order: relay A, relay D, relay B, relay C and relay E.

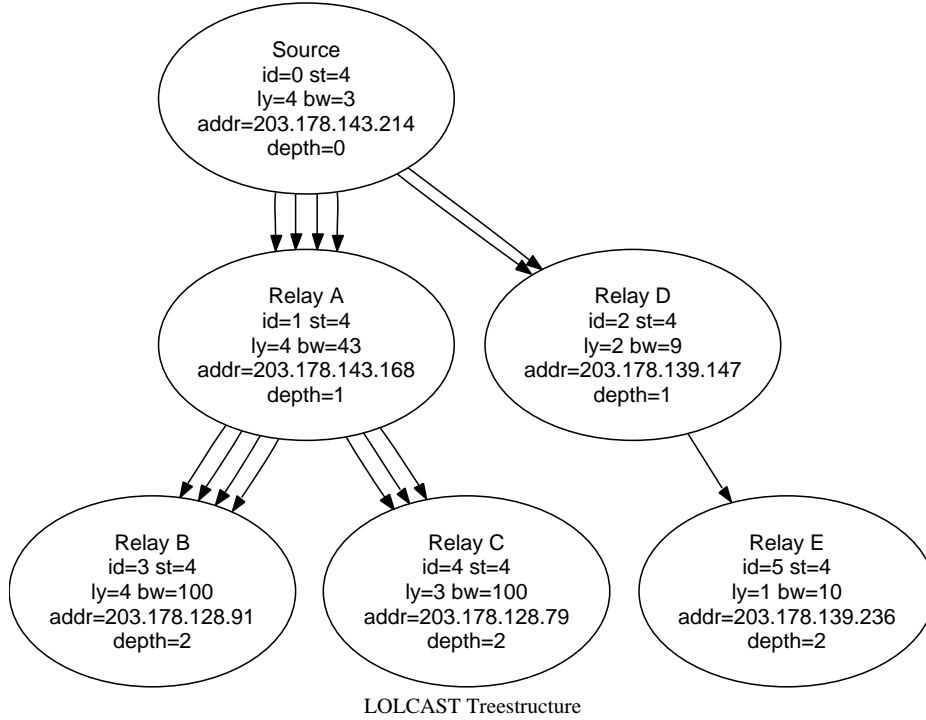


Figure 6.3: Generated LOLCAST Tree

The generated multicast tree is illustrated in Figure6.3. Figure shows the parameters of the node and the address. Number of path between each node indicates the number of layer it relays. Each node is receiving the data with their request and relaying it to other nodes. As the node receives data, bandwidth parameter is reduced by calculating the data bandwidth.

Figure6.4 shows the screenshot of running LOLCAST application in relay node A. Relay node A is receiving data with four layers. Video data with high quality, audio data and text data showing the information of the event is played all together. Information of current tree and data is offered by web page at source node. Relay node uses this information to select number of layers to receive. In addition, message log shows each of the message received from other nodes.

Figure6.5 illustrates example of message log at relay node E receiving Join Redirect message from source node. Join Redirect message includes parent candidate list which source generated. Message shows a list with four entries which first entry is relay node D. Relay node E joined to relay node D using this message as illustrated in Figure 6.3.

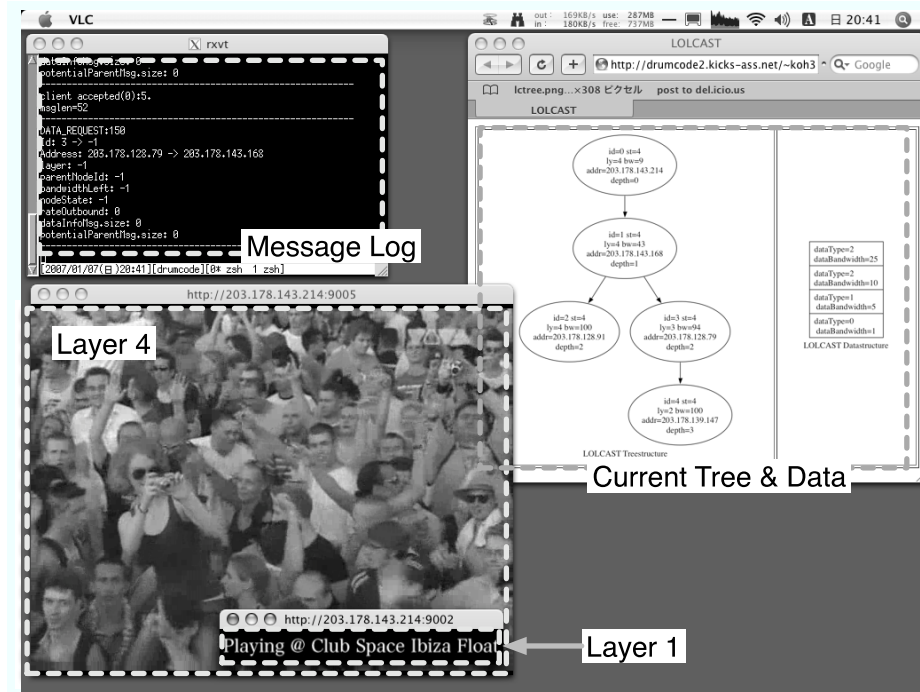


Figure 6.4: Screenshot at Relay Node A

```

-----
JOIN_REDIRECT:123
Id: 0 -> 5
Address: 203.178.143.214 -> 203.178.139.236
layer: -1
parentNodeId: -1
bandwidthLeft: -1
nodeState: -1
rateOutbound: 0
dataInfoMsg.size: 0
potentialParentMsg.size: 4
  nodeInfoMsg.0: nodeId= 2, address= 203.178.139.147, layer= 2
  nodeInfoMsg.1: nodeId= 4, address= 203.178.128.79, layer= 3
  nodeInfoMsg.2: nodeId= 0, address= 203.178.143.214, layer= 4
  nodeInfoMsg.3: nodeId= 1, address= 203.178.143.168, layer= 4
-----

```

Figure 6.5: Join Redirect message received at relay node E

6.2 Functional Comparison

In this section, functional comparison of LOLCAST with other Overlay Multicast protocols is done. Comparison between protocols are done by using the fundamental propositions stated in Section 1.3. There are two metrics, functions for adapting heterogeneity of end node and functions for adapting instability of end nodes. Table 6.3 shows the functional comparison between recently proposed protocols.

Functions of LOLCAST is compared with Narada [5, 8], HostCast [15], Okada's work [14] (A New Approach for the Construction of ALM Trees using Layered Video Coding), Yang's work [27] (A Proactive Approach to Reconstructing Overlay Multicast Trees) and PRM [20] (Probabilistic Resilient Multicast).

Table 6.3: Functional comparison of Overlay Multicast Protocols

Protocol	Node Heterogeneity	Tree and Node Instability
LOLCAST [17]	Abstract Layered Data Structure	Congestion avoidance Multi-path layer distribution
Narada [5, 8]	Multi-version	Reactive Approach
Okada's work [14]	Multi-layer	Proactive Approach (Backup parent candidates)
Koguchi's work [26]	Multi-layer	Reactive Approach
LION [18]	Multi-layer	Reactive Approach
PALS [19]	Multi-layer	Reactive Approach
HostCast [15]	None	Proactive Approach (Redundant data sending path)
PRM [20]	None	Proactive Approach (Randomized data forwarding)
Yang's work [27]	None	Proactive Approach (Pre-calculate backup node)

Functions for adapting heterogeneity of end node

For the metric, functions for adapting heterogeneity of end node, several protocols use multi-version or multi-layer approach. Narada [5, 8] uses multi-version approach, and Okada's work [14], Koguchi's work [26], LION [18] and PALS [19] uses multi-layer approach. As illustrated in Section 2.5.1 there are

advantages and drawbacks for both approaches. In addition, both approaches are supporting only video data. From this reason, user could not freely select the media type or quality of the content on demand and impossible to satisfy the research objective. In comparison, LOLCAST used abstract layered data to support this demand. Abstract layered data structure supports both fixed data used in multi-version approach and layer coded data used in multi-layer approach. In addition, combination of both approaches and other particular kind of data can be supported as illustrated in Section.3.1.

Functions for adapting instability of end node

Proposed functions for adapting instability of end node can be classified in to two types: reactive approach and proactive approach as stated in Section 2.5.2. Several protocols takes reactive approach: Narada [5, 8], Koguchi's work [26], LION [18] and PALS [19]. In contrast proactive approach is taken by Okada's work [14], HostCast [15], Yang's work [27] and PRM [20].

In HostCast, multiple path from the source to the node is prepared in advance. This backup paths to the source node runs through its primary parents grandparent node and uncle node. In case of node failure, node quickly changes to backup path to recover to the multicast tree. By preparing the backup path in advance in the control topology, node could find new parent node quickly. However, it still requires process time for switching the control topology and also to request new parent for the data. There is same issue in Okada's work and Yang's work that discontinuation of data fundamentally occurs for only constructing the path in control topology. In addition, Okada's work has another issue that there is a possibility of saved potential parent list to become invalid. PRM takes different approach by using the randomized forwarding method to construct a redundant path in data topology. However it has a large drawback in bandwidth utilization for sending overlapping data in a case such as live video streaming which is our target application.

Compared with recent approaches, LOLCAST takes two methods for adapting end node instability using the characteristics of abstract layered data structure illustrated in Section 3. LOLCAST proposed multi-path layer distribution method for fast recovery from node failure and congestion avoidance method in case of network congestion. In multi-path layer distribution method, node after joining to the primary parent with requesting layer, it constructs a backup path to other parent nodes on data topology. Specifically node receives data only with the base layer from multiple backup parent nodes. The difference between PRM [20] is lies in the data sent form backup nodes. By not sending the entire data redundantly, this method only sends

the base layer of the abstract layered data. Multi-path layer distribution methods offers fast recovery method compared to other proposed method by small drawback for sending base layer redundantly. Furthermore, proposed congestion avoidance method offers a countermeasure for network congestion. In case of network congestion, node sends a request to the parent to drop the sending number of layers to avoid it.

6.3 Performance of protocol process

This section represents the performance evaluation for LOLCAST using implemented LOLCAST application in simulation mode. In this simulation, load at source node is simulated by measuring the performance of Join and Leave Procedure. Objective for the simulation is to evaluate that ordinary user which does not have large resource environment can service to large group of people in a reasonable cost. This is one of our fundamental proposition stated in Section 1.3.

First, environment for the performance evaluation is illustrated. Next parameters set in LOLCAST application is illustrated briefly. Furthermore, the procedure for the performance evaluation is described. Next, performance of Join Procedure is evaluated by adding number of nodes to the multicast tree. Last, performance of Leave Procedure is evaluated by letting the node to leave from multicast the tree randomly.

6.3.1 Environment for performance evaluation

Hardware and software environment for the performance measurement is illustrated in Table 6.4. LOLCAST application is compiled with option “-O2” and run in single user mode for preventing unnecessary context switch.

Table 6.4: Hardware and software environment

CPU	Intel Core Duo 1.66Ghz
Memory	2GB
OS	MacOSX 10.4.8
Compiler	4.0.0 20041026 (Apple Computer, Inc. build 4061)
Compiler Option	-O2

6.3.2 Parameters for performance evaluation

This section represents the parameters used in the performance evaluation. Table 6.5 illustrates each of the parameter set in this evaluation. Bandwidth of each layer stored in layered data is fixed to 10; therefore data with 4 layers will have bandwidth of 40 for full layer. Outbound rate at source node is set to 1.5. For simplicity, backup rate at relay node is fixed to 0.2. Simultaneously, outbound and inbound bandwidth for relay node is fixed to 100. Number of layers served by the source node is set to 4 and the request from each node is set to random. Number of nodes joining to the tree is set to 10000.

Table 6.5: Parameters set for performance evaluation

Parameter	Value
Number of nodes	10000
Outbound rate (Source)	1.5
Backup rate (Relay)	0.2
Max number of layers	4
Requesting layer	random

6.3.3 Measurement procedure

Performance evaluation for LOLCAST is done in two steps. In the first step, nodes requesting random layers joins until it reaches the maximum number of nodes, which is set to 10000. In this period, process time at source node for running Join Procedure is measured for each node. After the first step finishes, nodes begin to leave from the multicast tree running Leave Procedure disorderly. Similarly, the process time at source node is measured for each node. When the tree becomes only with the source node, the evaluation finishes.

6.3.4 Measurement Result for Join Procedure

The main process done in Join Procedure is to find parent candidate list using the tree structure at source node as illustrated in Section 4.4.1. Figure 6.6 plots the process time of Join Procedure as the relay node size increases. In addition figure 6.6 illustrates the average process time compared with relay node size in the tree. The parameters are set to 4 for maximum layer

and request for the layer is randomized. Overhead view of graph shows that as the relay node size increases in the tree, process time for Join Procedure gets longer. Process time for the node is forming a distinguishing sets of four lines. This is due to the process done in *generatePCS()*. For not to burden the bandwidth left for a node relaying a lot of layers, *generatePCS()* searches the parent candidate from the node which relay layers close to the request as illustrated in Section 5.3.3. Consequently the search time differs from the requesting number of layers as illustrated in Figure 6.6. Another mentionable point is that each line shows an periodic increase in process time. This arises from the sort function called in *generatePCS()*. Detailed description for this phenomena is done below.

In this experiment, outbound bandwidth for relay node is fixed to 100. In consequence, node receiving four layers supports two child nodes and node receiving one layer supports ten child nodes. Accordingly, the cycle for depth increase is five times larger in requesting four layer than requesting one layer. In the sort function called in *generatePCS()*, each of parent candidate is compared and sorted by three conditions in the following order: number of layer, depth in tree and bandwidth left. This means that number of times for condition check decreases when depth gets larger. This is the explanation of why periodical increase of process time occurs and the cycle of increase differs between number of layers.

Table 6.6: Average process time for Join Procedure

Relay node size	Process Time (useq)
1000	249
5000	1177
10000	2411

6.3.5 Measurement Result for Leave Procedure

After Join Procedure completes protocol process, Leave Procedure is evaluated using the tree formed in join phase. The parameters are set to 4 for maximum layer and the size of relay node starts from 10000 until it reaches 1, only with the source node.

As described in Section 4.6, leave node is forced to wait for its child nodes to rejoin to the tree for preventing discontinuation of the service. Source node will run *generatePCSRejoin()* illustrated in Section 5.3.3 for each child node to find new parent node in the tree.

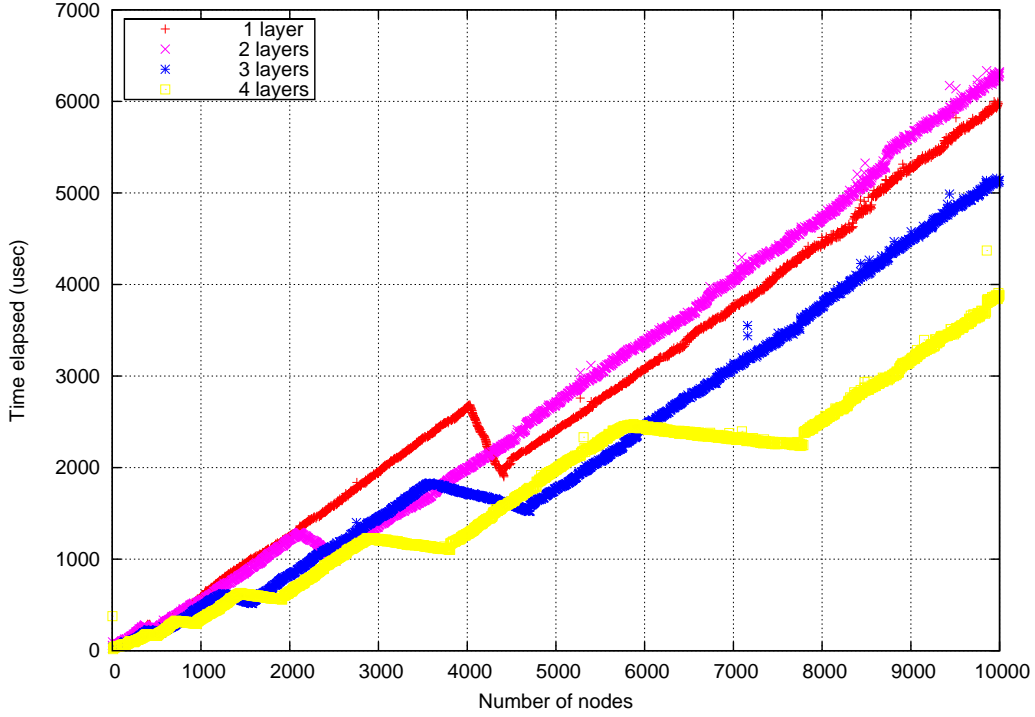


Figure 6.6: Process time for Join Procedure (4 layers/random layer/random leave)

This can be learned from the graph shown in Figure 6.7. Overhead view of the graph shows that as the relay node size decreases, process time for Leave Procedure shortens. Performance for Leave Procedure is divided into distinctive set of lines distinguished by the number of child nodes which leave node maintains. This is due to the rejoin process for child nodes. Leave node must wait for all of the child nodes to run *generatePCSRejoin()* and rejoin to the new parent. Altogether, process time will be multiplied by the number of child node it maintains. In addition, average process time for Leave Procedure is illustrated in 6.7.

In each set of lines separated by number of child nodes, process time differs and forming a line. The reason for this is same as in Join Procedure. Requesting number of layer of child nodes will affect the process time of *generatePCSRejoin()* for searching appropriate parent node. The line run along with the bottom of the graph is set of nodes which has no child node. The process for this case will only require searching the leave node from the tree structure and erasing it. In addition, from the reason that outbound bandwidth is fixed to 100, node with 10 child node is receiving and relaying one layer to other nodes.

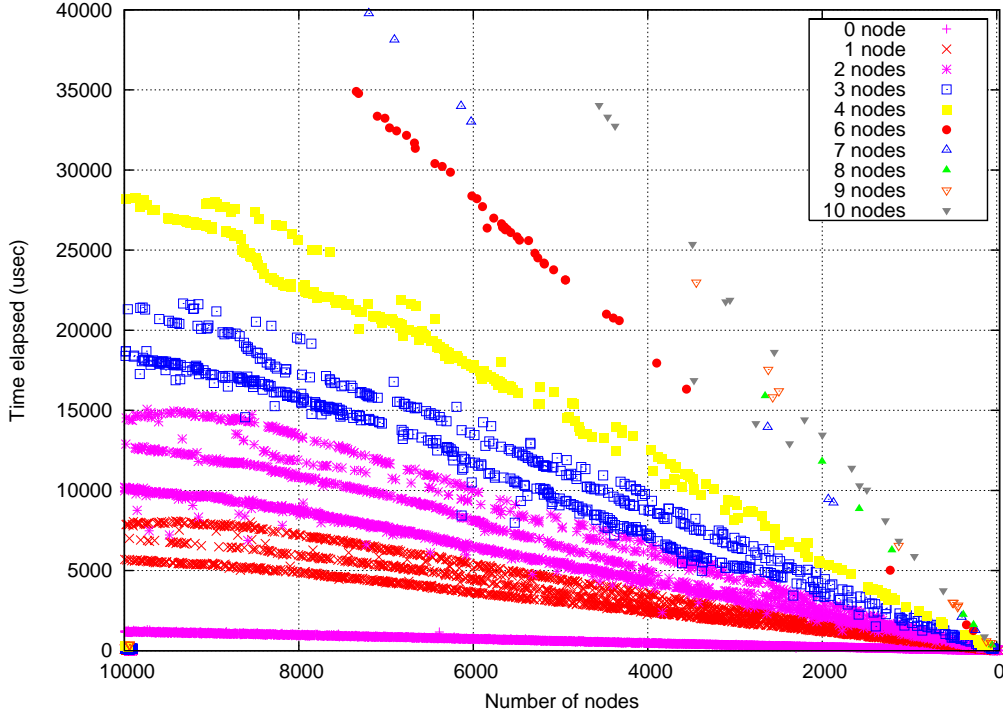


Figure 6.7: Process time for Leave Procedure (4 layers/random layer/random leave)

6.4 Summary

In this chapter illustrated the evaluation for LOLCAST. Evaluation for LOLCAST has been performed by three methods. First verification of protocol process has been confirmed by an experiment using LOLCAST application in a case of streaming live dance party event. The result show that source could serve a streaming service to number of nodes in a limited resource environment. Furthermore, each relay node freely selected media type of the content on demand by requesting the number of layer for the data.

Next, functional comparison with other proposed Overlay Multicast protocols has been done. Protocols has been compared with two metrics: functions for adapting heterogeneity of end node and functions for adapting end node instability. In the first metric, existing protocols support multi-layer or multi-version approach to support this. However both approach has issues left to support research objective as illustrated in Section 6.2. LOLCAST supports abstract layered data structure for adapting to the receivers various request for the data. In second metric, existing protocols takes reactive approach and proactive approach to support this. However both approach

Table 6.7: Average process time for Leave Procedure

Relay node size	Process Time (useq)
10000	7915
5000	6415
1000	4145

still need a time for reforming the control topology to receive the data. LOLCAST supports congestion avoidance method and multi-path layer distribution method adapting end node heterogeneity. Multi-path layer distribution method realize fast recovery method from node failure by small drawback in bandwidth. In addition, congestion avoidance method offers a counter-measure for network congestion using the characteristics of abstract layered data.

Last, performance of protocol process has been evaluated for verify that user could maintain the multicast tree with a ordinary resource environment. Result shows that process time for Join Procedure and Leave Procedure linearly increases as the group size get large. However the process time remains very low in our targeted group size, which is several hundreds.

According to the results illustrated above, LOLCAST meets the three fundamental propositions stated in Section 1.3: “User could send and receive contents with ordinary resource environment”, “User could freely select media type or quality of the content on demand” and “Seamless content delivery”.

Chapter 7

Conclusion and Future Work

7.1 Conclusion

This study focused on to support creative activity of ordinary Internet users distributing real-time streaming contents to large group of peoples. In this paper, we have introduced LOLCAST as an adaptive Overlay Multicast protocol for real-time group communication in a heterogeneous environment to realize this objective.

Recent group communication models had issues to accomplish our objective. Server-client model, CDN, IP Multicast can not support ordinary users from both technical and policy issues. Overlay Multicast can support ordinary user, but still had two issues to solve: adaptation to end node heterogeneity and adaptation to end node instability.

To solve the issues left in Overlay Multicast, this research proposed LOLCAST as an adaptive Overlay Multicast protocol for real-time group communication in a heterogeneous environment. Design of LOLCAST has been done including two distinguishing characteristics. First it supports abstract layered data structure for adapting to users heterogeneous resource environment and interest level to the content. Abstract layered data supports not only the existing multi-version and multi-layer data, but it can also support combined data from various media type abstractly. Number of layers in the data is used as the primary metric to construct the multicast tree. Second, LOLCAST has multi-path layer distribution method and congestion avoidance method for the solution to end node instability using the characteristics of abstract layered data. After the design, implementation of an streaming application using the LOLCAST protocol has been done.

Evaluation of LOLCAST has been done by three methods. Verification of the protocol process using the implemented application, functional com-

parison of LOLCAST with other Overlay Multicast protocols and performance measurement of the protocol process. Evaluation result shows that users could send and receive contents with ordinary resource environment and freely select media type or quality of the content on demand. Furthermore, LOLCAST prevents discontinuation of the content delivery in case of network congestion and multicast tree partition. Altogether, it has been confirmed that LOLCAST solved the major issues in Overlay Multicast research.

7.2 Future Work

For the future work, integration of multi-path layer distribution method and congestion avoidance method should be concerned to work as a complete protocol. Furthermore refining phase of the multicast tree should be considered in case of nodes requesting low layer burden the node serving high layer. Finally, we are planning to release the implemented application as an sample application for using LOLCAST protocol.

Bibliography

- [1] HITACHI. 360 度どこからでも見る事ができる立体映像ディスプレイ技術. <http://www.hitachi.co.jp/New/cnews/040224a.html>, 2003.
- [2] Bb@nifty: Bb clip. <http://bb.nifty.com/clip/movie/>.
- [3] Stephen E. Deering and David R. Cheriton. Multicast routing in datagram internetworks and extended lans. *ACM Trans. Comput. Syst.*, 8(2):85–110, 1990.
- [4] Christophe Diot and Brian Neil Levine and Bryan Lyles and Hassan Kassem and Doug Balensiefen. Deployment issues for the ip multicast service and architecture. In *IEEE Network Vol.14, num 1*, pages 78–88, 2000.
- [5] Yang hua Chu, Sanjay G. Rao, and Hui Zhang. A case for end system multicast (keynote address). In *Proceedings of the 2000 ACM SIGMETRICS international conference on Measurement and modeling of computer systems*, pages 1–12. ACM Press, 2000.
- [6] P. Francis. Yoid : Extending the internet multicast architecture. In *Technical report, AT&T Center for Internet Research at ICSI (ACIRI)*, April 2000.
- [7] B. Zhang, S. Jamin, and L. Zhang. Host multicast: A framework for delivering multicast to end users. In *IEEE Infocom*, 2002.
- [8] Yang Chu, Sanjay Rao, Srinivasan Seshan, and Hui Zhang. Enabling conferencing applications on the internet using an overlay multicast architecture. In *Proceedings of the 2001 conference on Applications, technologies, architectures, and protocols for computer communications*, pages 55–67. ACM Press, 2001.
- [9] Sylvia Ratnasamy, Paul Francis, Mark Handley, Richard Karp, and Scott Schenker. A scalable content-addressable network. In *Proceedings of the 2001 conference on Applications, technologies, architectures,*

BIBLIOGRAPHY

- and protocols for computer communications*, pages 161–172. ACM Press, 2001.
- [10] Suman Banerjee, Bobby Bhattacharjee, and Christopher Kommareddy. Scalable application layer multicast. In *Proceedings of the 2002 conference on Applications, technologies, architectures, and protocols for computer communications*, pages 205–217. ACM Press, 2002.
 - [11] Duc A. Tran, Kien A. Hua, and Tai T. Do. Scalable media streaming in large peer-to-peer networks. In *Proceedings of the tenth ACM international conference on Multimedia*, pages 247–250. ACM Press, 2002.
 - [12] Yatin Chawathe. Scattercast: an adaptable broadcast distribution framework. *Multimedia Syst.*, 9(1):104–118, 2003.
 - [13] John Jannotti, David K. Gifford, M. Frans Kaashoek, and James W. O’Toole Jr. Overcast: Reliable multicasting with an overlay network. In *5th Symposium on Operating System Design and Implementation (OSDI)*, December 2000.
 - [14] Yohei Okada, Masato Oguro, Jiro Katto, and Sakae Okubo. A new approach for the construction of alm trees using layered video coding. In *P2PMMS’05: Proceedings of the ACM workshop on Advances in peer-to-peer multimedia streaming*, pages 59–68, New York, NY, USA, 2005. ACM Press.
 - [15] Zhi Li and Prasant Mohapatra. Hostcast: A new overlay multicasting protocol. In *IEEE International Communications Conference (ICC)*, 2003.
 - [16] Dimitris Pendarakis, Sherlia Shi, Dinesh Verma, and Marcel Waldvogel. ALMI: An application level multicast infrastructure. In *Proceedings of the 3rd USNIX Symposium on Internet Technologies and Systems (USITS ’01)*, pages 49–60, San Francisco, CA, USA, March 2001.
 - [17] Kohei Ogura, Hideaki Imaizumi, Nakamura Osamu, and Jun Murai. Overlay multicast protocol for delivering hierarchical structured data. In *12th Workshop on Distributed Processing System (SIG-DPS)*, December 2004.
 - [18] Jin Zhao, Fan Yang, Quian Zhang, Zhensheng Zhang, and Fuyan Zhang. Lion: Layered overlay multicast with network coding. *IEEE Transactions on Multimedia*, 8(5):1021–1032, 2006.

BIBLIOGRAPHY

- [19] Reza Rejaie and Antonio Ortega. Pals: Peer-to-peer adaptive layered streaming. 2003.
- [20] Suman Banerjee, Seungjoon Lee, Bobby Bhattacharjee, and Aravind Srinivasan. Resilient multicast using overlays. *SIGMETRICS Perform. Eval. Rev.*, 31(1):102–113, 2003.
- [21] S. Banerjee and B. Bhattacharjee. A comparative study of application layer multicast protocols. 2002.
- [22] (ISO/IEC 13818-2). Mpeg-2 generic coding of moving pictures and associated audio information. 1995.
- [23] (ISO/IEC 14496-2). Mpeg-4 generic coding of moving pictures and associated audio information. 1999.
- [24] Guy Cote, Berna Erol, Michael Gallant, and Faouzi Kossentini. H.263+: Video coding at low bit rates. *IEEE Transactions on circuits and systems for video technology*, 8(7):849–866, Nov 1998.
- [25] Mengyao Ma, Oscar C. Au, and S.-H. Garry Chan. Multiple-description coding for error-resilient video transmission. volume 4, pages 1426–1431, October 2005.
- [26] Atushi Koguchi and Hidenori Nakazato and Hideyoshi Tominaga. A Tree Routing Method on Multi-Tree Application Level Multicast Streaming System. 6 2005.
- [27] Mengkun Yang and Zongming Fei. A proactive approach to reconstructing overlay multicast trees. *INFOCOM 2004. Twenty-third Annual Joint Conference of the IEEE Computer and Communications Societies*, vol.4, no.pp. 2743- 2753 vol.4, March 2004.
- [28] Sally Floyd, Mark Handley, Jitendra Padhye, and Joerg Widmer. Equation-based congestion control for unicast applications. In *Proceedings of the conference on Applications, Technologies, Architectures, and Protocols for Computer Communication*, pages 43–56. ACM Press, 2000.